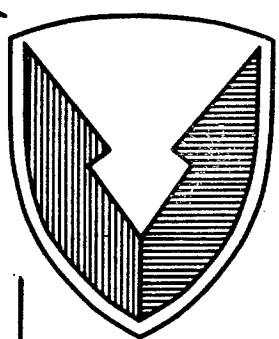
RD&E CENTER

Technical Report



No. 13463

ROBOTIC MOBILITY SYSTEM: ROAD FOLLOWING ALGORITHMS

Contract Number DAAE07-84-R-138

January 1990

Environmental Research Institute of Michigan
P. O. Box 8618
Ann Arbor, MI 48107
By ERIM Report No. 179100-40-F

Approved for Public Release:
Distribution is Unlimited

U.S. ARMY TANK-AUTOMOTIVE COMMAND
RESEARCH, DEVELOPMENT & ENGINEERING CENTER
Warren, Michigan 48397-5000

NOTICES

This report is not to be construed as an official Department of the Army position.

Mention of any trade names or manufacturers in this report shall not be construed as an official endorsement or approval of such products or companies by the U.S. Government.

Destroy this report when it is no longer needed. Do not return it to the originator.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for Public Release: Distribution Is Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBERS(S): 179100-40-F	5. MONITORING ORGANIZATION REPORT NUMBER(S): 13463
6a. NAME OF PERFORMING ORGANIZATION: Environmental Research Institute of Michigan; 6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION: U.S. Army Tank-Automotive Command
6c. ADDRESS (City, State, and ZIP Code): P.O. Box 8618, Ann Arbor, MI 48107	7b. ADDRESS (City, State, and ZIP Code): Warren, MI 48397-5000
8a. NAME OF FUNDING /SPONSORING ORGANIZATION: U.S. Army TACOM; 8b. OFFICE SYMBOL (if applicable): AMSTA-RRT	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAE07-84-R-138
8c. ADDRESS (City, State, and ZIP Code): Warren, MI 48397-5000	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Robotic Mobility System: Road Following Algorithms

12. PERSONAL AUTHOR(S)
Staff Report

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 9/84 TO 12/88	January 1990	140

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Image processing, vision sensor, planners, lasers, vehicle control, robotics

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Image processing algorithms were developed to permit use of a laser-based vision device for autonomous navigation of a wheeled vehicle. Imagery and processing results are given.

Operation of the laser-based vision device (ERIM's Autonomous Land Vehicle sensor) is described, including a new output signal that is sensitive to scene texture or vertical structure; other signals provide range and reflectance information.

Simulation software packages were developed to model a robotic vehicle's performance and to study advanced planner concepts required for flexible vehicle behavior. Plans to develop a demonstration vehicle are given.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Larry Sieh	22b. TELEPHONE (Include Area Code): (313) 574-5440	22c. OFFICE SYMBOL: AMSTA-RRT

PREFACE

The contract with the U.S. Army Tank-Automotive Command for "Development of Road-Following Algorithm," No. DAAE07-84-C-R138, was started in September 1984 and was completed in December 1988. The work was performed in the Electro-Optics Systems Laboratory of the Sensor Systems Division under the direction of Max Bair, Laboratory Director. Program managers were Dr. Robert Franklin during the first 2 years and Dwayne C. Carmer for the remainder of the program.

The sponsor's technical representative was Francis B. Hoogterp until November 1987, when Larry Sieh became COTR. Modifications were made to the contract to meet sponsor objectives and funding limitations. No-cost time extensions were granted to permit ERIM to modify its 3-D laser sensor and collect additional imagery.

The algorithms discussed in this final report and the original plan to demonstrate them on a robotic mobility vehicle system are intended to support the Army's robotic vehicle development efforts.

The following ERIM staff members are recognized for valuable contributions to this project and for contributing material for this report: R. Franklin, D. Carmer, F. Pont, N. Finzel, L. Harmon, J. Beyer, J. Trenkle and R. Watts.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1.0. INTRODUCTION

This report discusses autonomous land navigation. Robotic mobility for military vehicles would provide resupply and combat vehicles with capabilities to enhance our ground forces' effectiveness. This report shows that laser-based vision sensors provide the information needed for land navigation and could be used in either autonomous or semi-autonomous navigation modes.

2.0. OBJECTIVES

The objectives of the program described here deal with the development of computer software for autonomous guidance of a wheeled vehicle. Mission planning, image processing, and vehicle control all involve special software. The focus of this effort is on planning and image processing for data derived from a laser-based vision sensor.

Objectives of this contract were initially directed toward the development of road-following algorithms using 3-D laser sensor range data and assessment of the feasibility of using the laser sensor as the vision sensor for a robotic mobility vehicle. Based on ERIM's recommendation, the contract objectives were expanded to include the design, development, integration, and demonstration of an autonomous vehicle using the 3-D laser sensor, road finding and road following algorithms, and planner/navigator algorithms for autonomous navigation along road networks with and without obstacles.

A Chevrolet Suburban vehicle and remote control actuator components were acquired, but incremental funding was stopped and all work on the vehicle was stopped by a contract modification due to sponsor funding priorities. The emphasis was changed to provide for the documentation and delivery of the image processing algorithms and the collection of texture imagery using ERIM's autonomous land vehicle (ALV) 3-D laser sensor modified for this purpose.

3.0. CONCLUSIONS

Algorithms for processing imagery produced by ERIM's 3-D laser sensor have been developed and demonstrated for road following applications. With these algorithms, a robotic vehicle could be directed to stay on a road and avoid obstacles by detecting: (1) the geometric features of ditches or banks along the edges; (2) the smoothness of the road surface; (3) the vertical features of obstacles and highly textured off-road areas; (4) the reflectance

difference of the road and bordering areas; or, any one of the above. In many cases the geometric features from the range image are adequate, but, when these fail, the reflectance or texture images provide the information.

The range sensor, the range image processing algorithms, several vehicle models, and a vehicle control algorithm have been combined with a simulated road network to provide a dynamic autonomous vehicle navigation environment that runs on a VAX computer. This simulation package has been documented (see Appendix A) and a magnetic tape copy of the software provided to the sponsor. Vehicle control strategies and hardware design issues related to the image sensor, motion, and angle sensors may be investigated with this simulation software.

The general concept of the planner envisioned for the vehicle is described in Appendix B. Software was developed to demonstrate the use of a blackboard architecture for a flexible planner that addressed concurrent processing, specific information requests, and monitoring of exceptional situations. This software, developed on a Symbolics computer, is fully documented in Appendix C.

The plan for development of a demonstration robotic vehicle is described. It includes an inexpensive, commercial, 4-wheel drive vehicle, ERIM's 3-D laser sensor for vision data, and ERIM's laboratory computer system. A telemetry system would have linked the vehicle with the laboratory computer. The vehicle, a Chevrolet Suburban, and special motors and controllers were purchased but no hardware modifications were made because of sponsor-directed changes to the contract objectives. These purchased items were delivered to the sponsor.

4.0. RECOMMENDATIONS

Recommendations for further research and development involve the image processing algorithms, the vehicle control algorithms, the planner, and the vehicle. The image processing algorithms work on the imagery tested, but true robustness requires extensive testing with a wider variety of road types. Also, to be of greater use to the Army, algorithms for off-road situations should be addressed. The vehicle control algorithm should be changed from the pursuit strategy now used to an optimal preview control method based on a human driver model by C. MacAdam of the University of Michigan Transportation Research Institute.[1] Probably the greatest need is in the design of the planner to handle the

uncertainties inherent in real autonomous vehicle operations. The capabilities for a blackboard-based planner demonstrated in software must be improved as indicated in Section 5.4 and expanded to permit flexible robotic behavior.

Although the use of a commercial vehicle would be ideal for a contractor test-bed, it is now recognized that a military vehicle, such as the HMMWV, is probably best for reasons of commonality and demonstrations involving other vehicles in military scenarios.

5.0. DISCUSSION

5.1. General

The ALV program at Martin Marietta, Denver, funded by the Defense Advanced Research Projects Agency (DARPA), is using a laser-based vision sensor but only for obstacle detection. ERIM's approach has been to use the laser sensor for all vision functions and not rely on a color TV camera for finding the road edges, identifying landmarks or determining surface types. The laser sensor may be used to accomplish these same goals and is also capable of providing slope and surface roughness information without reliance on sunlight.

Although they are not addressed here, it is important to note that many issues unrelated to vision sensors or image processing arise when one considers autonomous land navigation. Is it necessary and feasible to build such a flexible on-board planner that all reasonable situations are handled autonomously, or is a semi-autonomous vehicle concept more appropriate as a development goal because in such a system the man-machine interfaces could be worked out to incorporate man's decisions/inputs at critical points in a mission? Of course, computer power needed in the vehicle, data transmission to the human operator, computer power needed at the control center and control signal transmission back to the vehicle all need further study from a systems viewpoint. Fully autonomous vehicles will have to be given the ability to cooperate with friendly forces and other robotic vehicles. If not, then a semi-autonomous mode could be provided to inject human control when cooperation is necessary. All of these issues could be grouped under a heading of "graded autonomy." It appears that the future development of ALVs will depend on resolution of such issues. This report shows that laser-based vision sensors provide the information one needs for land navigation and could be used in either autonomous or semi-autonomous navigation modes.

One of the key technologies for autonomous land navigation is the vision and image processing subsystems that provide information on the local scene for vehicle guidance decisions. With adequate local scene information, a vehicle may guide itself around obstacles and along roads or cross country along navigable routes. One vision device that is capable of providing local scene information is the 3-D laser imager first built by ERIM and demonstrated on the Adaptive Suspension Vehicle program at Ohio State.[2] That imager, as well as similar ones built by ERIM and provided to Carnegie-Mellon University and Martin Marietta, Denver, provides range information to every point in its field of view. The resulting array of image points may be interpreted to determine geometrical features of the scene and thus locate rocks, holes, trees, ditches, etc. The same imager also produces scene reflectance information that is registered with the range data and is useful for discrimination of scene elements having the same geometry but differing reflectance (or colors). As discussed in Section 5.2, a third signal may also be produced from the laser imager that is sensitive to the orientation of planar surfaces and to edges of objects; these signals are made into an image called a texture image that provides further information about the local scene features.

5.2. Laser Sensor

The basic laser sensor, its operating parameters, and modifications performed to produce the new texture signal output are presented here as well as examples of imagery, both raw and processed, for detection of road edges and obstacles. The basic image processing algorithm is also described. A report is referenced for additional details on the processing algorithms.

5.2.1. Description. The laser sensor used on this program was developed under DARPA funding for application to autonomous vehicle navigation. The sensor ERIM has now is similar to ones built and delivered to Ohio State University, Carnegie-Mellon University, and Martin Marietta. A photograph of the sensor head is shown in Figure 5-1; a set of low-voltage power supplies and the signal processing electronics are each packaged separately in rack-mounted units 14 inches high. A block diagram of the basic sensor is given in Figure 5-2. The dashed lines show the new signal paths and components added to produce the texture signal output. Sensor specifications are given in Table 5-1.

The basic 3-D sensor consists of a scanning mechanism which directs the laser beam and field of view of the detector to the scene. The modulator driver provides a modulated light

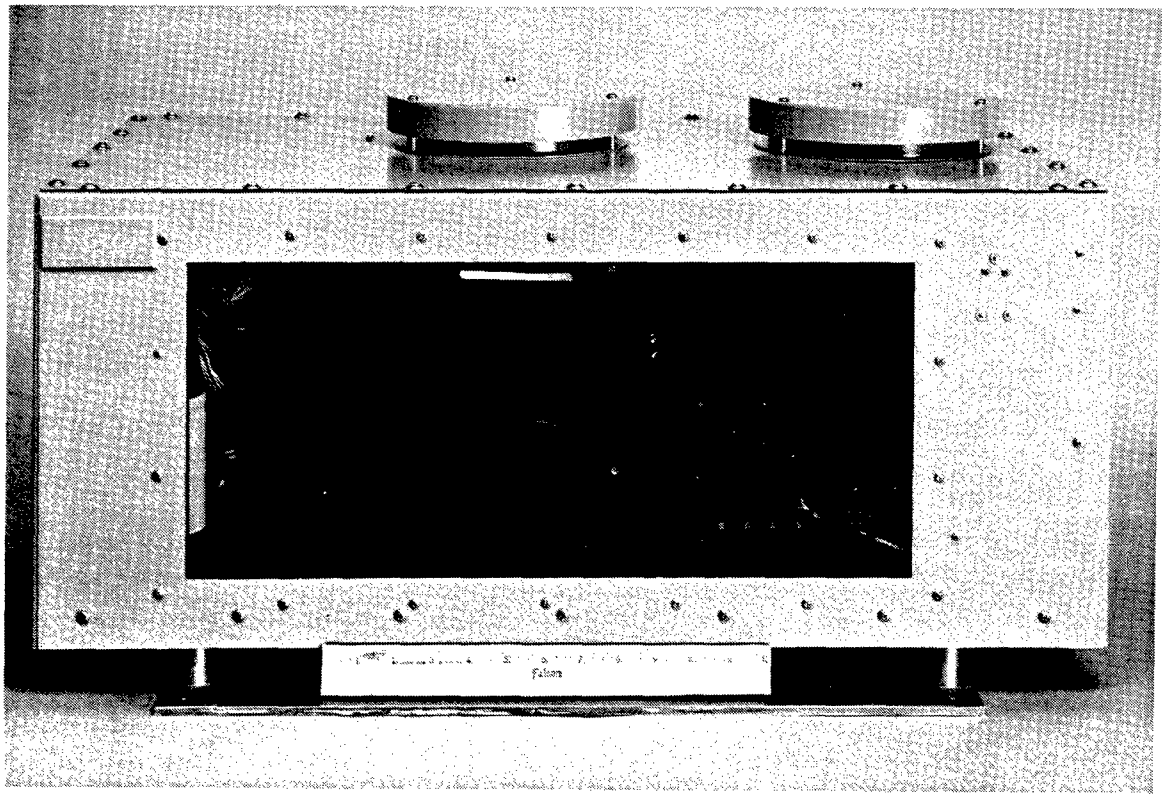


Figure 5-1. Laser-Based Vision Sensor. Output signals include range, reflectance and texture.

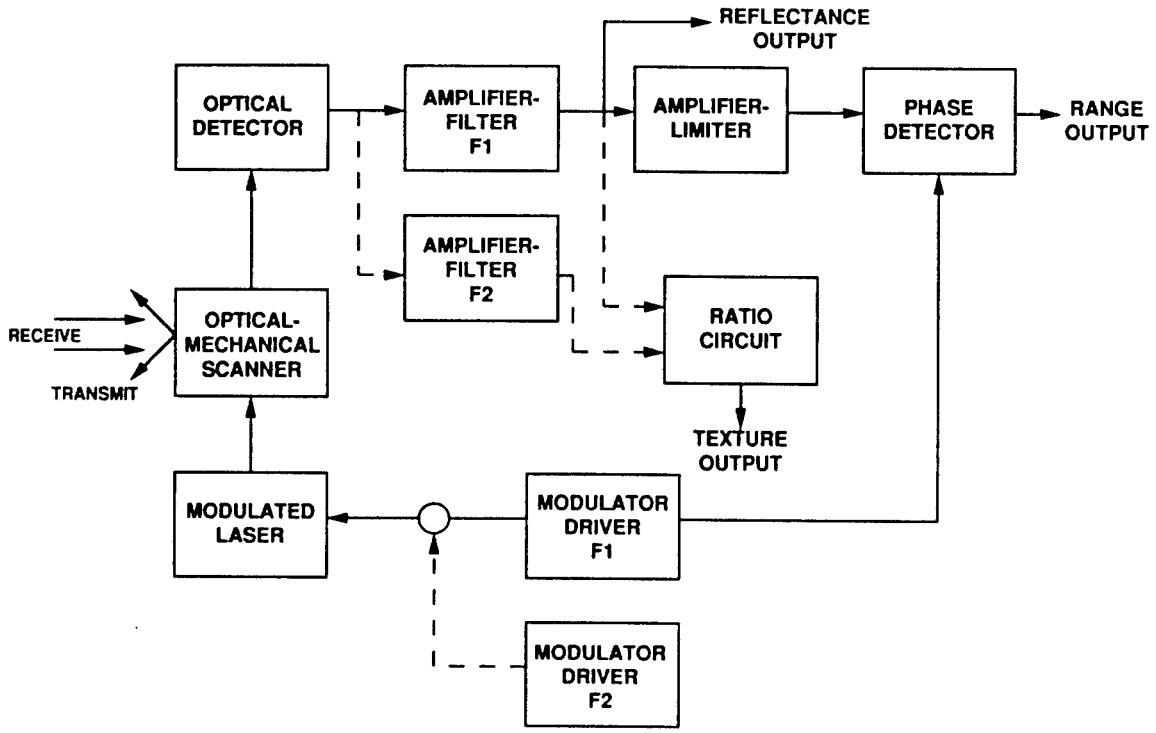


Figure 5-2. Laser Sensor Block Diagram with Texture Output

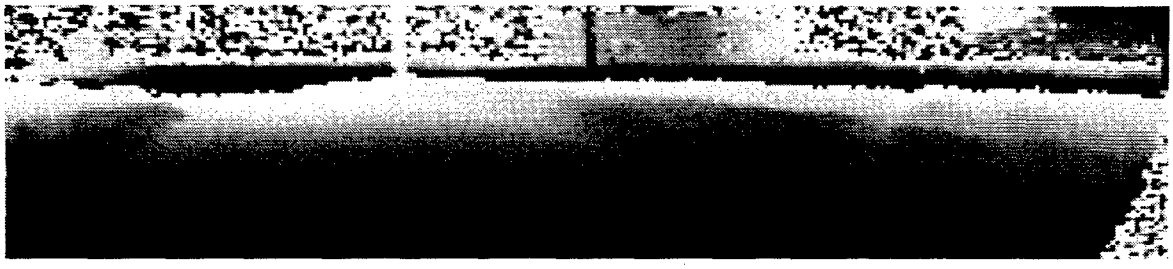
Table 5-1. Sensor Specifications

Frame Rate	2 per second
Frame Scan	Top to Bottom
Data Rate	92,160 Hz
Field of Regard (Deg)	
Vertical	30 Depression
Horizontal	±40 Measured from Center-line of Sensor
IFOV (Deg)	0.5
Range Resolution (ft)	0.25 (7.62 cm)
Range Noise (ft)	0.4 (12 cm) 10% Target Reflectance in Bright Sunlight
Ambiguity Interval (ft)	
For Range Data	64 (19.4 m)
For Texture Data	5 (1.524 m)
Wavelength (μm)	0.82
Output Format	8 Bits of Range 8 Bits of Log Reflectance or Texture

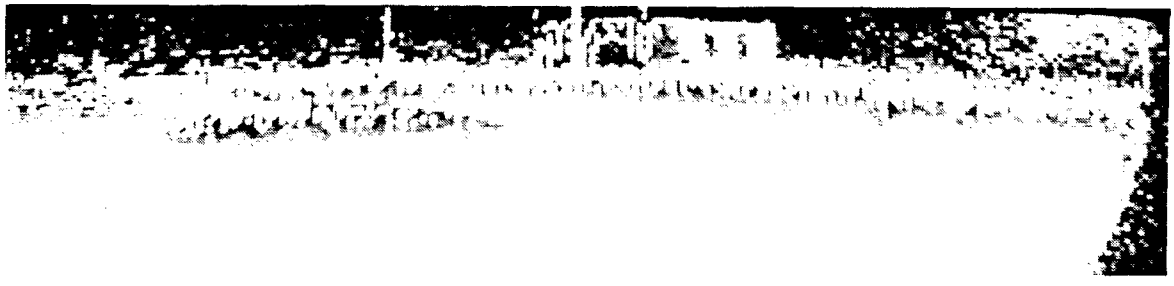
source via the laser and also a phase reference signal to the phase detector. The optical detector converts the reflected, modulated, optical energy to an electrical signal which is filtered and amplified, and its amplitude is outputted as a reflectance signal. It also passes into an amplifier-limiter to remove amplitude variations caused by the varying reflectance of the scene. The limited signal contains phase but no amplitude information and is the other input to the phase detector. The output of the phase detector is the phase difference between the reference and reflected signals and corresponds to the range from the sensor to the target. A modulation frequency of 7 MHz is used for the range data, resulting in an ambiguity interval of 64 ft.

The texture signal output is derived by modulating the laser at 100 MHz and taking the ratio of the amplitudes for the 100 and the 7 MHz received signals. This ratio decreases for flat earth areas in the scene but remains high when vertical surfaces are scanned. Thus, highly textured regions are bright in the texture signal and horizontal smooth areas are dark. The 100-MHz signal corresponds to a range ambiguity of 5 ft and thus, if the range values present within the laser beam's footprint on the ground have a dispersion approaching 5 ft, the modulated component in the optical detector's output is nearly cancelled by interference of the various phases present. The decrease in this 100-MHz signal causes the texture signal to decrease. If the surfaces have a low value of reflectance, both the 100-MHz and the 7-MHz signals decrease and the ratio is unaffected. The 7-MHz signal may be used as a normalizing factor since its range ambiguity is 64 ft and a dispersion of 5 ft has little effect on its magnitude. Since the texture signal is a function of the range dispersion within the laser beam's footprint, it will depend on the laser's divergence angle, the range, and the height of the sensor as well as the properties of the scene. Range dispersion of some degree is likely to be present when the laser beam intercepts both the edge of an object and some background material. Thus, in a texture image the edges are likely to be dark.

5.2.2. Example Images. Range and reflectance images for a road and building are illustrated in Figure 5-3. These are raw data in angle, angle, range coordinates (top) and angle, angle, amplitude coordinates (bottom). A plan view of the range image is shown in Figure 5-4 where the coordinates are down range, cross range, and elevation. An intermediate step in the algorithm for locating the road edges is shown in Figure 5-5 where highly rough areas are bright and smooth areas are darker. Figure 5-6 has boundary lines placed to



(a) Laser Range



(b) Reflectance

Figure 5-3. Laser Range and Reflectance Images of a Road and Building

Figure 5-4. Plan View of Range Image from Figure 5-3(a)

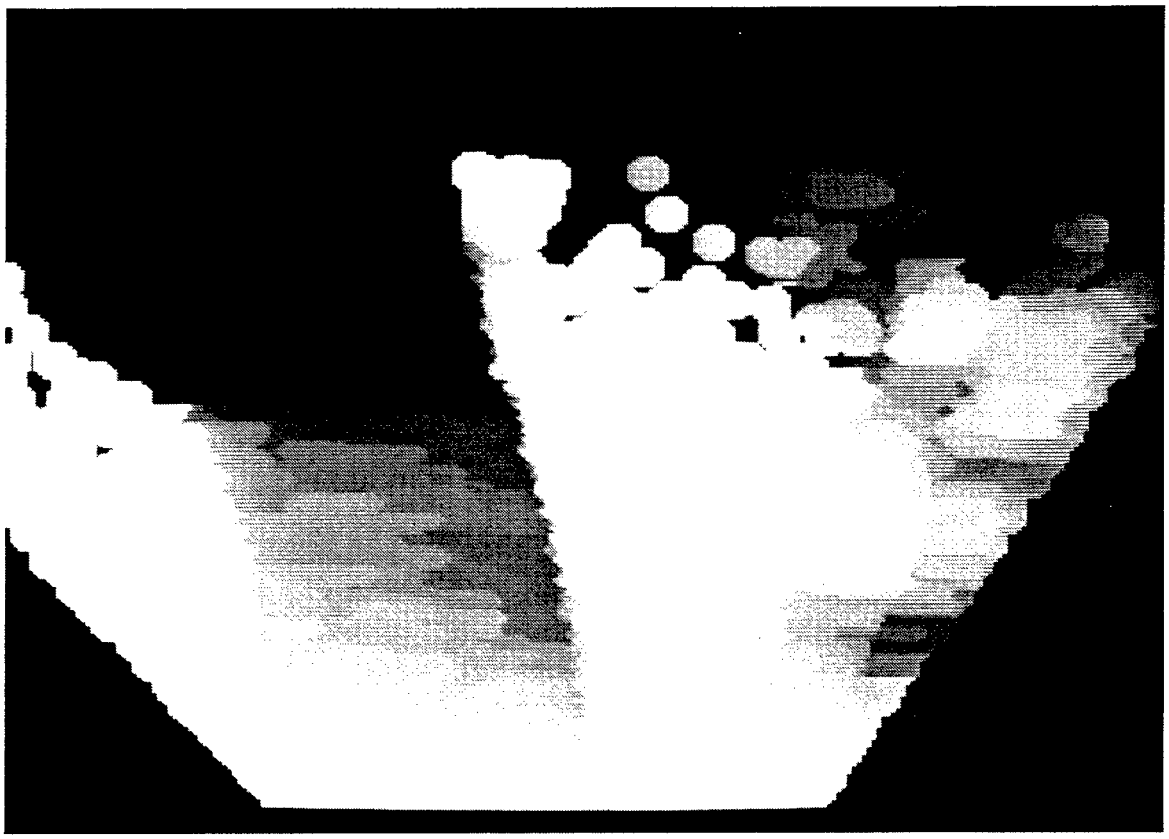


Figure 5-5. Partially Processed Range Image from Figure 5-3(a)

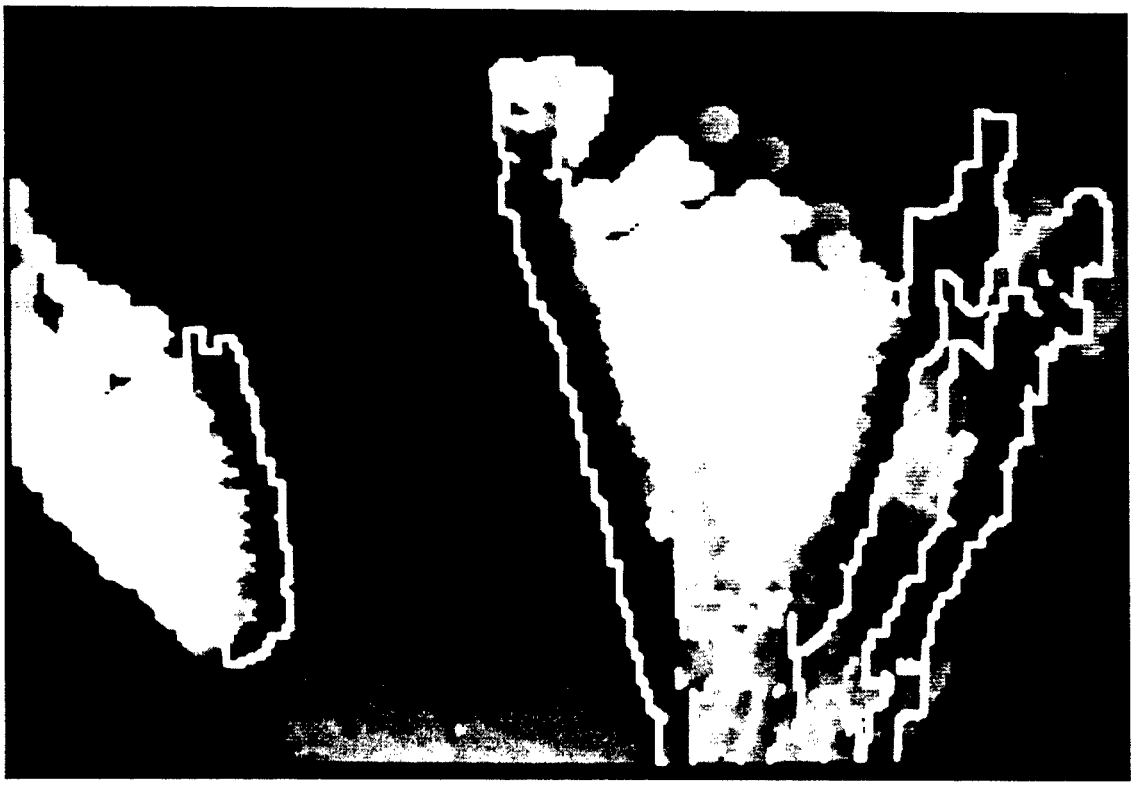


Figure 5-6. Identification of Road Edges with Range Data

outline the smoother areas. In this image, the flat road was found, on the left, but two paths (tire tracks) were also located on the right.

Another set of range and reflectance imagery is shown in Figure 5-7 where an obstacle, a car, is on the edge of the road. Figure 5-8 shows the processed image with a center-line path indicated for obstacle avoidance.

Another example of range and reflectance imagery is given in Figure 5-9. The top image is raw range data, the middle one is raw reflectance and the bottom one is range-corrected reflectance. This corrected reflectance image was processed to remove the effects of range, that is, the laser return signal falls off as the square of the range causing all features to take on darker shades of gray with increased range; when corrected, the resulting image of the road depends only on reflectance values and is independent of range (i.e., the road should appear uniformly dark from the top to the bottom of the image). The corrected reflectance image was used to find the road edge for the cases shown in Figure 5-10 where the range data failed to reveal the left-hand edge.

Examples of raw range and texture imagery are given in Figure 5-11. An intersection is on the left, an open road ahead, and a curb, grass, and tree trunk on the right. Plan views of the data are shown in Figure 5-12. Highly textured areas are outlined, and the center of the road is marked in Figure 5-13.

5.2.3. Image Processing Algorithms. The road following algorithms used to process these example images take advantage of the fact that roads are generally smoother than the surrounding terrain. The range data algorithm includes the following steps:

- Flag the bad pixels. If the reflectance signal is too low, the range data for that pixel will be noisy and unreliable; pixels of that type are flagged and not used in the range data processing.

- Resolve the unambiguous range values. Since the 3-D sensor has an ambiguity interval of 64 ft, the range values start repeating for distances greater than 64 ft from the sensor. Any uncertainty is resolved by depending on the bottom scan line of the image, the one covering ground closest to the vehicle, to be true range. Thus, by screening the range values from the bottom to the top of the image,

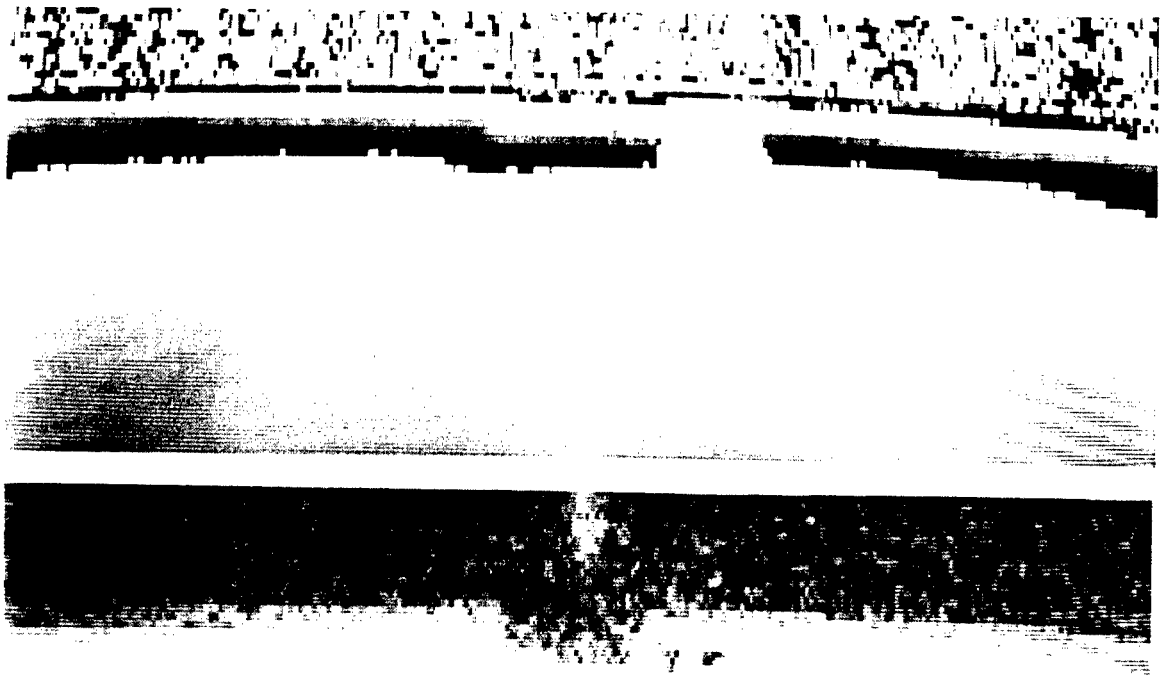


Figure 5-7. Laser Range (Top) and Reflectance (Bottom) Images of Road and Car

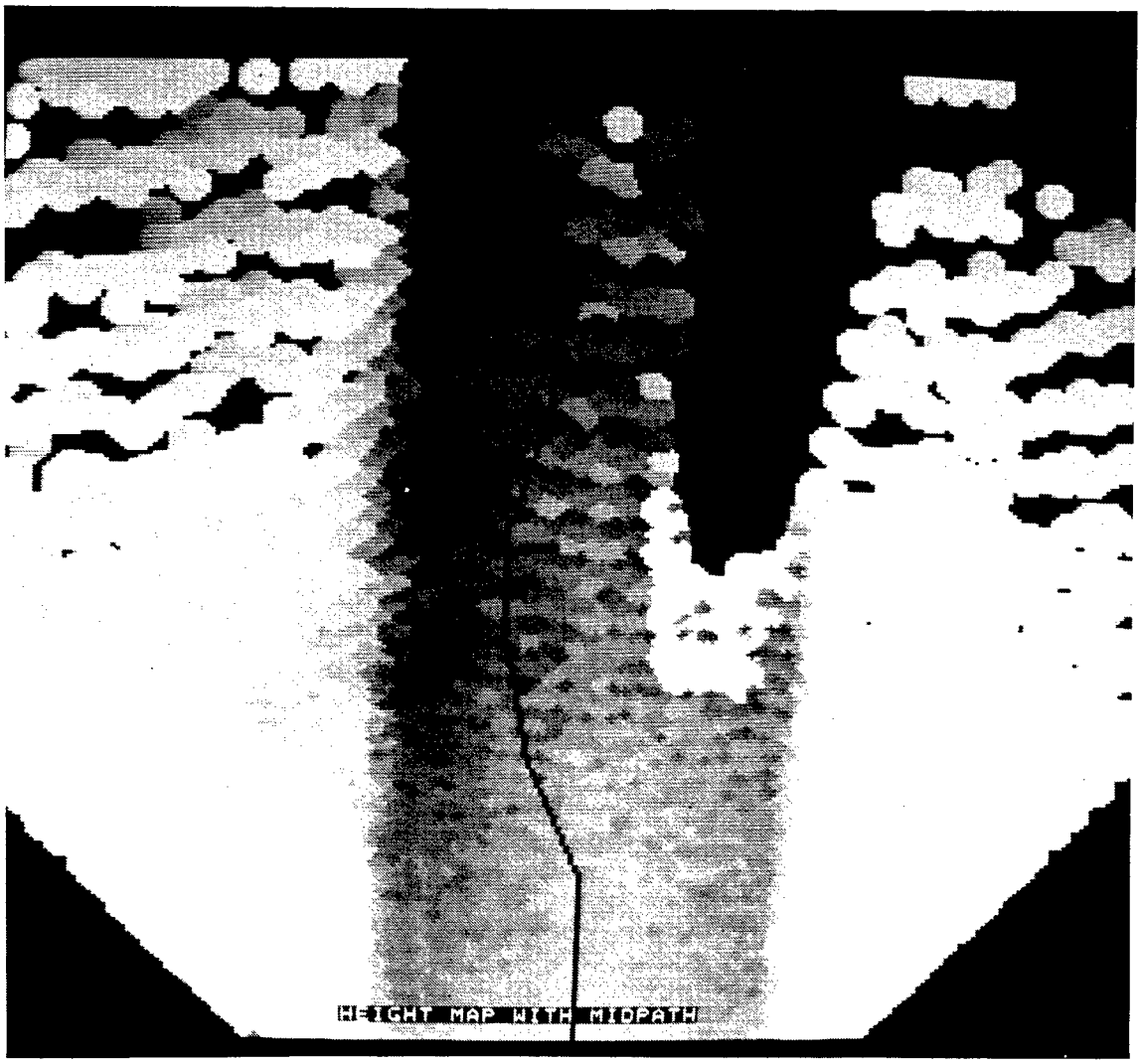


Figure 5-8. Range Image Processed for Obstacle Detection and Avoidance

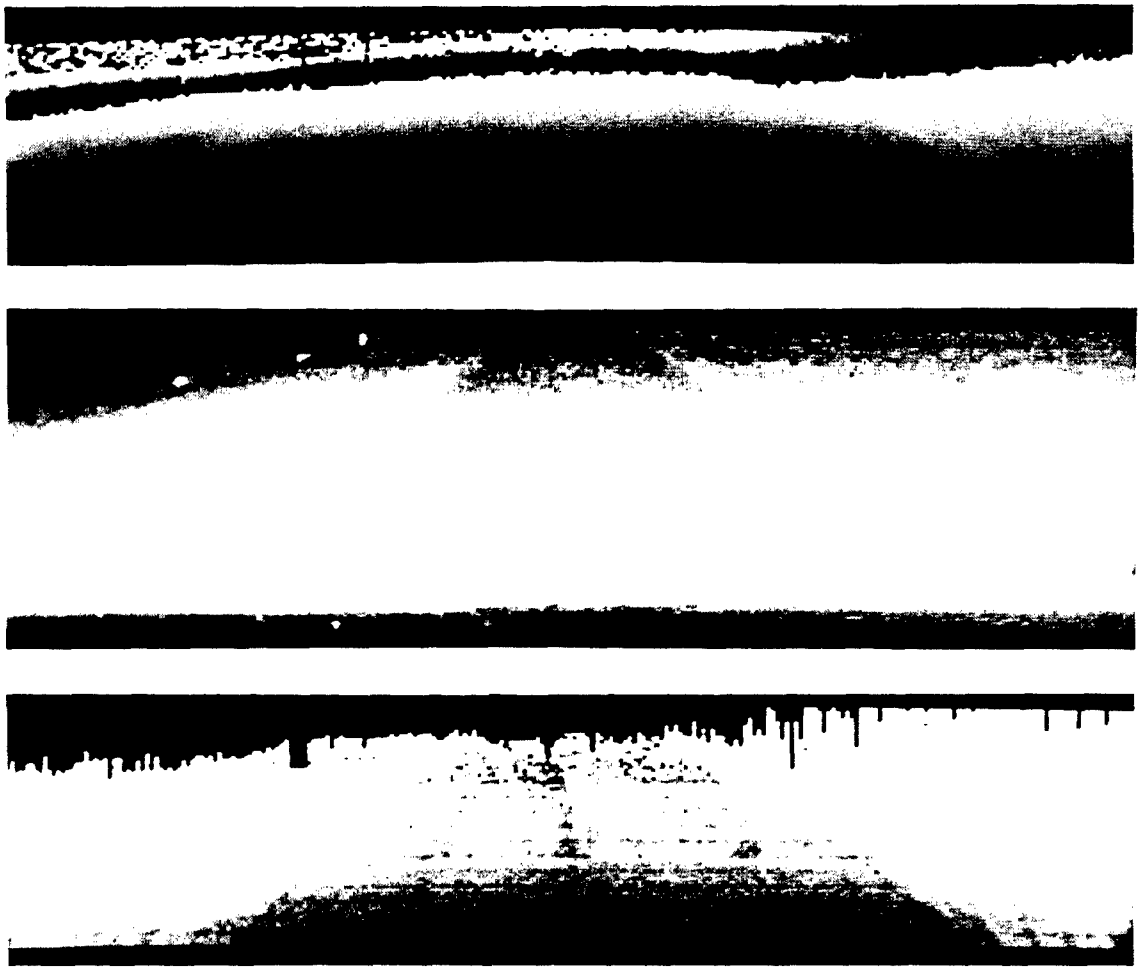
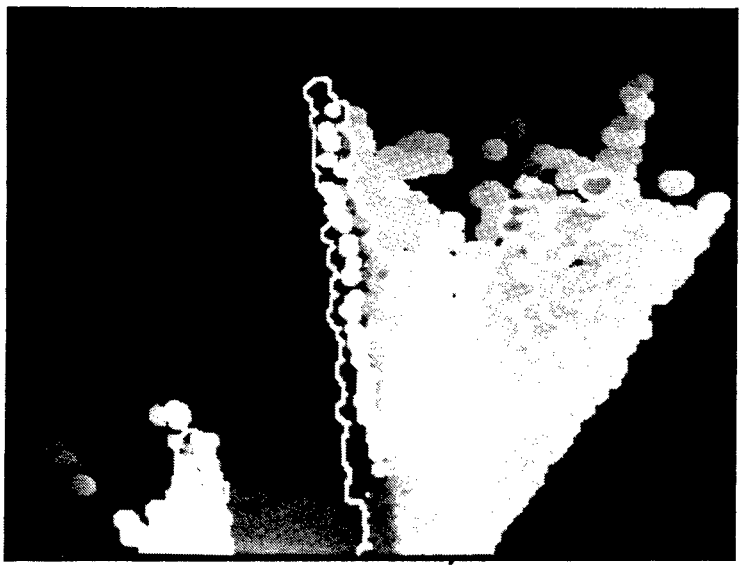
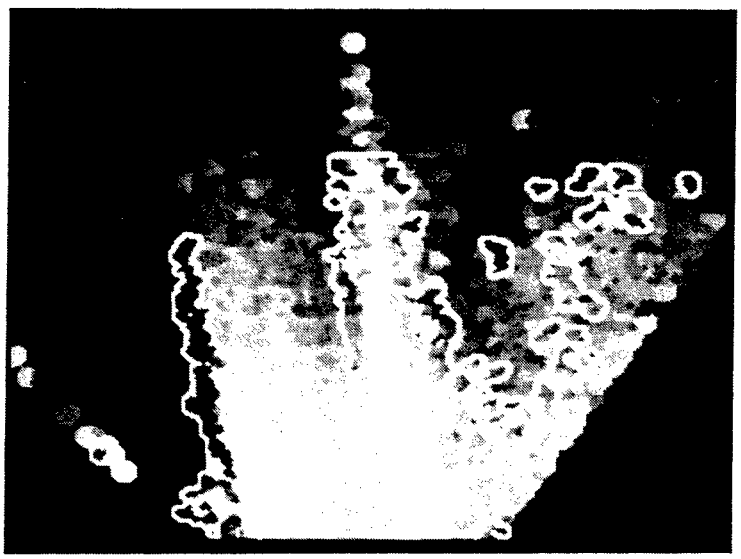


Figure 5-9. Range, Reflectance, and Range-Corrected Reflectance Imagery (Top to Bottom)



(a) Range



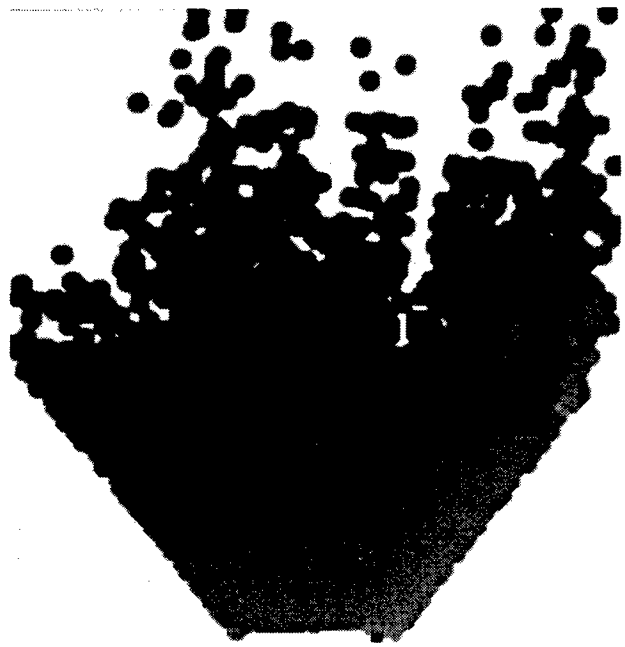
(b) Reflectance

Figure 5-10. Combined Use of Range and Reflectance Imagery

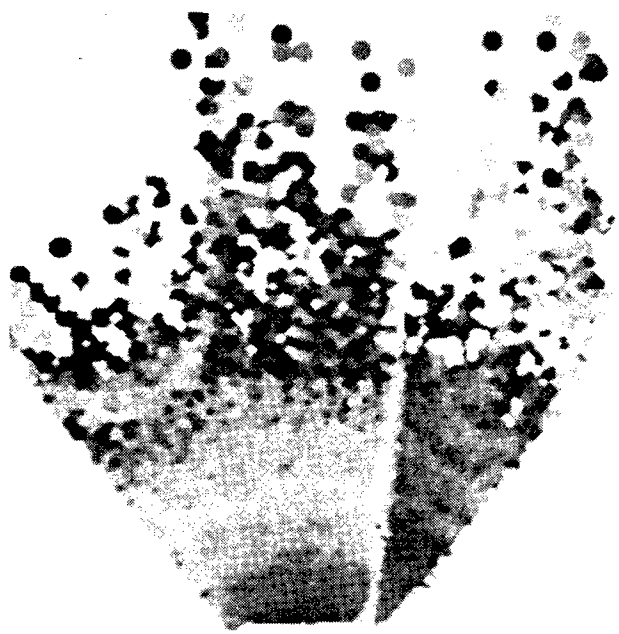
(a) Range

(b) Texture

Figure 5-11. Range (a) and Texture (b) Imagery



(a) Range



(b) Texture

Figure 5-12. Plan Views of Range (a) and Texture (b) Imagery

Figure 5-13. Road Edges and Centerline Marked for Image from Figure 5-11(b)

the place where the second ambiguity region starts is located and 64 ft are added to all values from that point until the third ambiguity is reached, where, if the data are reliable, 128 ft are added. Note, the images seldom enter the third ambiguity region because of system noise levels; the top lines of the images are random noise values because of weak returns.

- Coordinate conversion. The raw data, in azimuth, elevation, and range coordinates, are converted to rectangular coordinates for ease of the further processing steps and to obtain a map-like format for use in vehicle navigation. In the plan view image, height is referenced to a flat earth.

- Nearest neighbor resampling. After coordinate conversion, some pixels have zero height values because of bad pixels flagged in the first step. Neighboring values within a four-pixel region are processed and a value is assigned to fill in the bad pixels with estimated heights.

- Cross-range texture. Smooth and nonsmooth areas, e.g., curbs, ditches, and potholes, are determined on a pixel basis by passing structuring elements of varying sizes over the elevation image. Thresholds of texture height may then be applied to segment the image into areas of weak (low threshold) or strong (high) edges.

- Connect weak and strong edges. Weak-edge pixels are connected to strong ones if they are within a 20-pixel distance so as to restore continuity to the processed image. Other weak edges are discarded, and the result is a map delineating the boundaries of smooth and nonsmooth areas.

Texture images are processed in a similar way except that the pixels are already in units that represent texture. Reflectance images are also processed similarly except that one must depend on the road being lesser or more reflecting than the surrounding area; this boundary, if it exists, is then marked as the road edge. It is expected that roads will always be discernable in one or more channels of data from the laser sensor, that is, in the range, reflectance or texture images.

More details on the 3-D laser sensor, the data it produces, and the algorithms used to process those data are given in a 117-page report published by ERIM in December 1987 entitled "Range and Reflectance Processing Workshop."[3] The processing steps are fully illustrated with color-coded images. Test results from an effort to process the imagery in real-time, as would be needed to guide a vehicle when the sensor produced 2 frames per second, are also presented in the workshop report. That test revealed the need to modify the computer and algorithms to increase the speed by a factor of 4; methods to achieve this increase were identified.

5.3. Autonomous Land Vehicle Simulation

The consensus among people familiar with the ALV is that hardware issues are more easily addressed in a simulation mode than in real hardware tests. Thus, an ALV simulation was developed to address system design questions. A block diagram of the simulation is given in Figure 5-14; elements of the simulation are listed in Table 5-2. This simulation runs on the VAX computer. Its inputs are: (1) a geometric world of the road, its surface features, edges, and obstacles; (2) the vehicle, including its initial location, heading, and velocity; (3) the 3-D laser sensor and its mounting height and depression angle; (4) sampling rates, digital resolutions, and random errors to be used for the range image and the odometer, steering angle, and heading sensors; and (5) the route to be followed toward a destination.

The simulation was designed to provide outputs for the study of the steering and velocity control system, the accuracy of image reconstruction, and how well a specified path could be followed, i.e., the control loop could be closed and the results studied. The basic vehicle model was provided by the sponsor and produced roll and pitch angles as a function of vehicle steering and acceleration commands as the tires encountered height profiles on the road surface. The original model represented the M151 Jeep and was modified to include the Chevrolet Suburban and HMMWV vehicles also. In addition, the model software was revised to work at a simulated time increment of 1/160 second since the range sensor generates an image at the rate of 160 lines per second. The image generated by the simulated sensor was reconstructed to account for motion of the vehicle and then it was processed by the algorithms illustrated in Section 5.2 to locate the road edges and find the centerline of the clear path around obstacles.

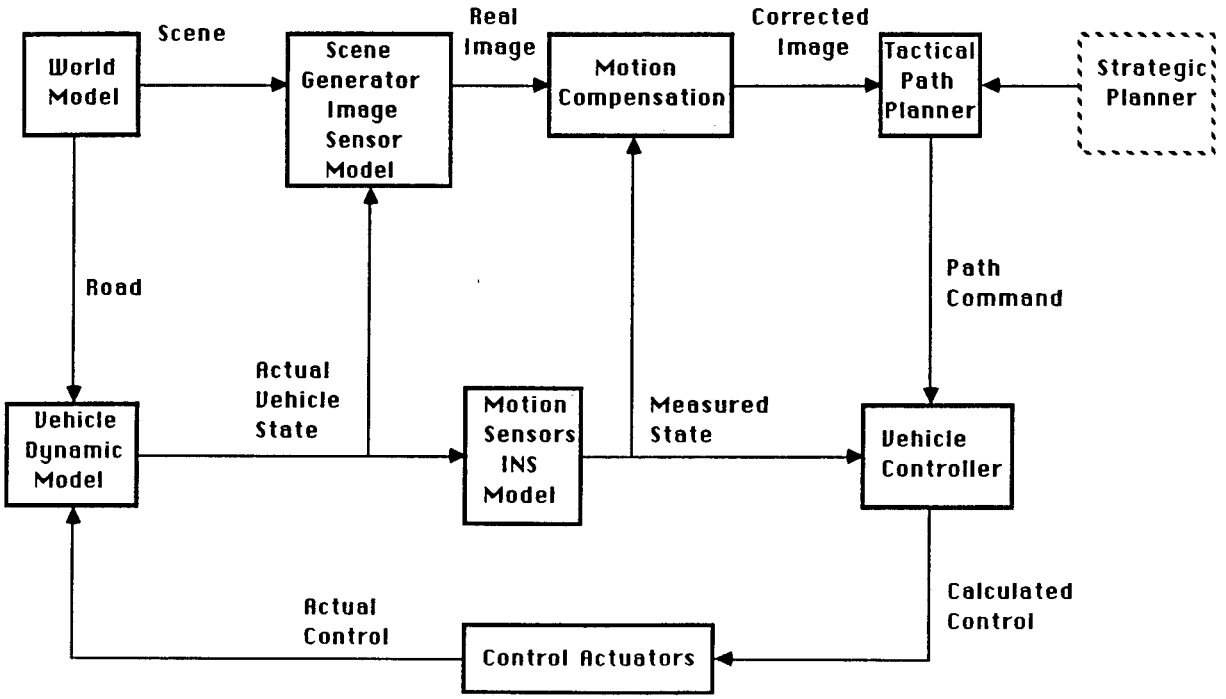


Figure 5-14. Dynamic Vehicle Simulation

Table 5-2. Dynamic Vehicle Simulation

1. World Models
   (a) Straight Roads with 90° Intersections
   (b) Curved Roads
   (c) Roughness Factors Included

2. Scene Generator (3-D Sensor)
   (a) Ray Trace (No Mixed Pixels)
   (b) Back Trace (No Mixed Pixels)
   (c) Roll, Pitch, Heading, Velocity, Position from Vehicle

3. Motion Compensation
   (a) Down Range Motion
   (b) Change of Heading (~)

4. Tactical Path Planner
   (a) Path in Vehicle Coordinates
   (b) (0,0) Represents Vehicle at Start of Frame
   (c) Uses Range Data
   (d) Obstacle Avoidance Included

5. Vehicle Dynamic Models
   (a) HMMWV
   (b) Jeep M151
   (c) Chevrolet Suburban

6. Motion Sensor Models
   (a) Accuracy (Number of Bits)
   (b) Noise (Mean and Variance of Error)
   (c) Update Rate

7. Vehicle Controller
   (a) Based on Tangent Circles (Initial Only)
   (b) Could Be Replaced with Model of Human Driver

8. Actuator
   (a) Force Model (Initial)
   (b) Could Be Replaced with Physical Models

A description of the elements of the ALV simulation is given in Appendix A where three types of roads are illustrated. The results of one test case are given in Figure 5-15 where the vehicle path (centerline and tire paths) is shown as it avoids obstacles on the road, traveling from left to right.

The software for this ALV simulation has been documented and provided to the sponsor for use on a VAX computer.

5.4. Planner Simulation Software

The planning system envisioned for the autonomous vehicle is based on a flow of information derived from vision sensor images (range, reflectance and/or texture) and the merging of those local data with global information (maps) and mission or goal information. As discussed in Section 5.2, the image data are converted to a map format to better relate to vehicle routing as well as to global data in map format. The envisioned planner concept is described in Appendix B.

The concept presented in Appendix B includes a discussion of the tactical, strategic, reactive, and integrative planning activities that are required. Effective planning for autonomous mobility is extensive and computationally demanding. The best software architecture for the planner is believed to be one based on a blackboard system. Various blackboard architectures, as defined in the literature, are described and references are given in Appendix B. A basic blackboard architecture consists of three components: knowledge sources for extraction of various types of information; a blackboard for handling requests for information and the results; and a control structure to handle priorities and allow for concurrent processing activities.

To work out implementation details for the planner, a software package was developed to demonstrate a limited set of functions. A Symbolics computer was used for the Blackboard Architecture demonstration as described in Appendix C. A listing of the software is included as part of that appendix. The demonstration accepted a set of real range images as input data. Characteristics of these images included a section of straight road, a curve, and a wide open area. These images were made available to the system as fast as they could be read from the disk memory in order to simulate continuous sensing. Once into the blackboard architecture (BBA), each image set in motion the following default-driven processing sequence:

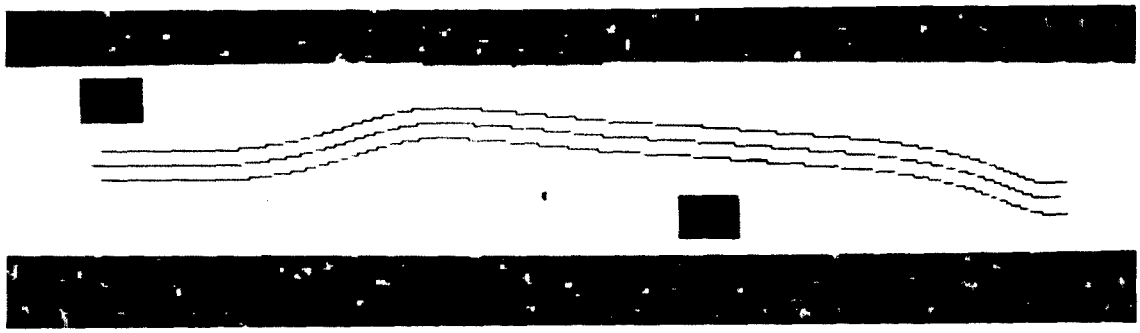


Figure 5-15. Vehicle Simulation Output for Obstacle Avoidance

- Convert the sensor image to plan-view (x y z)
- Delimit the usable portion of the image
- Mark points in the image which appear to be edges
- Extrapolate a line, if possible, for the left side of the road
- Extrapolate a line, is possible, for the right side of the road
- Notify the user if the left road side cannot be found
- Notify the user if the right road side cannot be found

Unless other input to the system occurred, such as a user-supplied request for a particular result, these steps proceeded in a sequential, data-driven fashion for each image as it entered the system.

The potential flexibility of the BBA was demonstrated by two distinct means. First, although each image was processed sequentially in the default, data-driven mode, nothing prevented processing of more than one image at a time. Demonstration of concurrent processing of several images, in both similar and different stages of processing, was demonstrated.

Second, the use of requests was shown. Requests could be entered at the keyboard for any of the data producible by the above processing steps. If the requested datum could not be determined due to lack of input data, the request would propagate through the system, i.e., the process invoked by the original request would post requests within the BBA for the input data it required. At the same time, the system was still processing other sensor images in its data-driven default mode.

Thus, the final BBA demonstration showed the capabilities of the system to handle real data, to process data concurrently (whether data-driven, request-driven, or both), to propagate requests internally when required, and to monitor for exceptional situations (loss of one or both road sides).

Data organization in the BBA developed for TACOM is consistent with those reported in the literature. A hierarchical data structure was not implemented, but data were subdivided

and grouped based on level of abstraction and control/domain considerations. Our implementation of knowledge sources may be unique; insufficient detail is available in literature descriptions to be sure. Our implementation consists of checker and actor components, as do most of those documented. No trigger has been implemented, but pre-conditions exist above and beyond input data requirements. The control structure implemented is similar to the most basic of those found in the literature. More extensively developed control structures have been documented, but no globally acceptable one is known to exist. The control is very much dependent on the specific problem, causing each BBA's control structure to vary considerably.

There are many potential areas of future work and possible improvements on the TACOM BBA. As previously mentioned, the control structure is an area of research itself. No attention has been given to issues of concurrency--interference between knowledge sources. This must be addressed, as the knowledge source structure makes parallel processing an inherent part of the BBA. The request mechanism of our BBA requires improvement; no allowance has been made for "hanging requests"; if a request cannot be answered, it remains in a busy wait, using precious system time. The failure to provide requested data must be recognized and propagated through the system to the originating requestor.

Another primary issue, that of time, has not been adequately handled in our system. Time plays a very important role in the validity of sensory information and results derived from these data. Time also is an issue in emergency situations, where a less costly (in terms of time to solution) approach to a decision may be necessary to avoid disaster.

These are some of the principal outstanding problems with the current BBA. It is hoped that future efforts could be directed to improvements in these areas.

5.5. Demonstration Vehicle

Plans were made to develop a commercial four-wheel drive vehicle into a robotic vehicle to demonstrate the utility of the laser sensor and image processing algorithms for the road following application. However, because of sponsor-directed contract changes, these plans were not implemented. The vehicle, a Chevrolet Suburban with a 6.2-liter diesel engine, was purchased. Materials to fabricate vehicle controls were also purchased. This section of the report will provide an overview of the vehicle development plan.

Actuators were to be installed in the vehicle to accomplish computer control of steering, shifting, accelerating, and braking functions. Vision information would be provided from ERIM's 3-D laser sensor mounted on the vehicle's roof. A telemetry system would transmit the digital images to the laboratory-based computer system where image processing would provide the information needed to generate vehicle commands in accordance with mission goals and planner strategies. These high-level commands would be transmitted to the vehicle where the on-board computer would interpret them and issue appropriate control signals to the actuators. Thus, the control loop would be closed and mobility along road networks and around obstacles would be achieved. Joystick controllers installed in the vehicle and in the laboratory would be available to allow for system test and control by a human operator. Also, a vehicle driver would be in place at all times and could override the computer controls for safety reasons if necessary.

Block diagrams of the vehicle and laboratory-based instrumentation are shown in Figures 5-16 and 5-17, respectively. The TV camera shown in Figure 5-16 would be used only for system checkout and for visual assessment of the operations; the TV imagery would not be processed for any control information. The telemetry system was considered necessary to allow cost effective use of both special and general-purpose computers provided in the laboratory by ERIM. Fully autonomous robotic vehicles would obviously require all data processing and control functions to be performed by on-board computers specially adapted and packaged for that purpose. The telemetry system would also offer more versatility in the testing of algorithms using real imagery provided by the vehicle-mounted laser sensor.

The primary element of the laboratory computer system, Figure 5-17, is the CYTO-HSS pipelined neighborhood processor. It is the specialized image processor that is used to extract feature information from the laser sensor data. An improved method of coding the algorithm instruction sets was identified to allow processing the images in real time, i.e., at a rate of 2 frames per second. Specifications were developed and bids received for the telemetry system indicated in Figures 5-16 and 5-17; work stopped before any orders were placed. Directional antennas and provisions for manual or automatic tracking were considered necessary to avoid signal quality problems caused by multipath (interference) propagation of the microwave signal. A line-of-sight range of 1.5 miles was specified. If the channel frequencies were in the 1710 to 1850 MHz military band, a band used by many teleoperated test vehicles now, then there would be compatibility with other test facilities such as

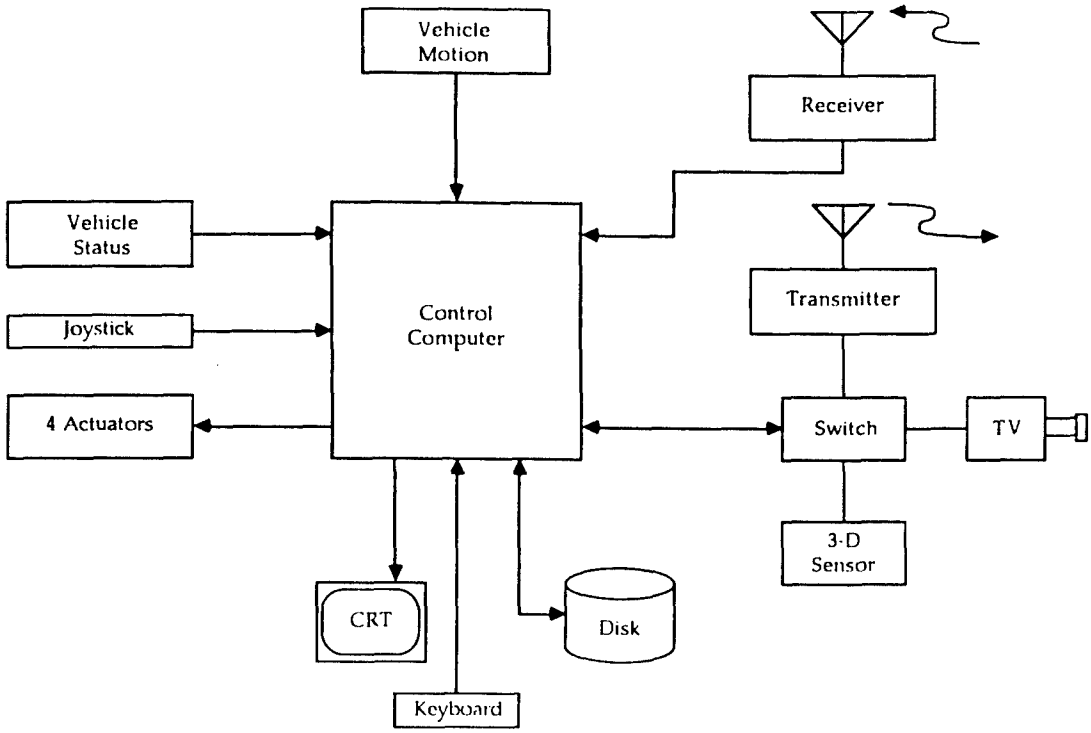


Figure 5-16. Vehicle Control

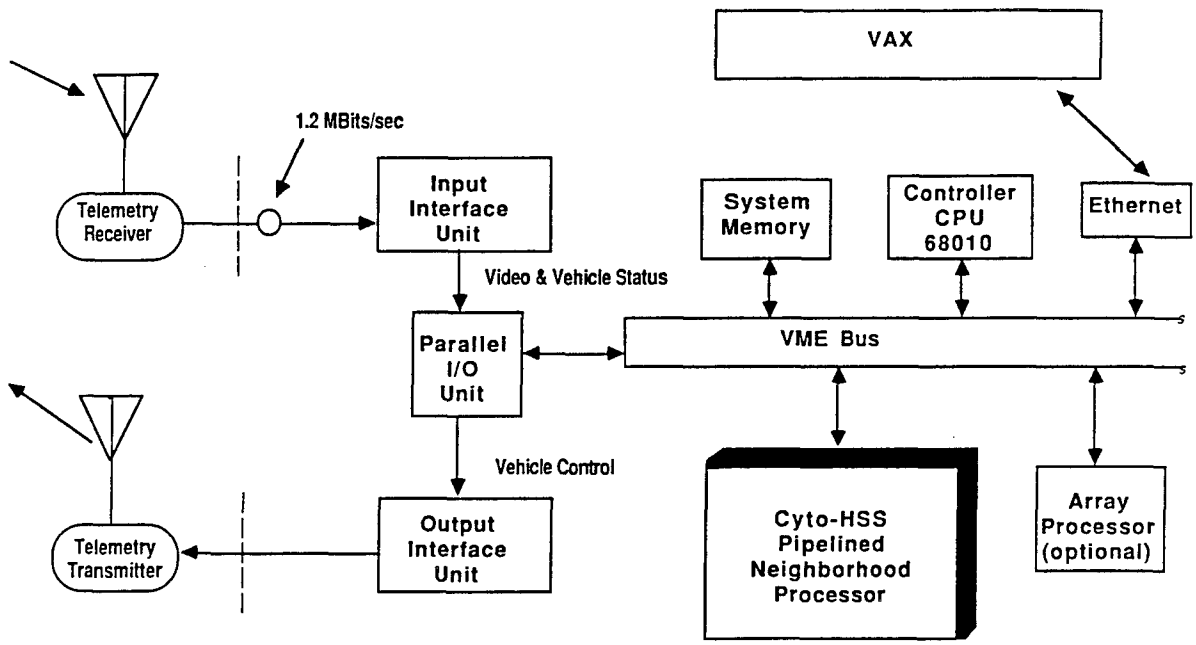


Figure 5-17. TACOM Mobile Vehicle Control Base Processor

Martin Marietta. Although the command signal channel could operate with a low bandwidth, it was tentatively decided to use two identical transmit-receivers with 5 MHz bandwidth (maximum) to accommodate TV signals. The laser imagery itself requires only 400 KHz bandwidth for each type (range, reflectance, and texture). Thus, the data link would have excess capacity.

Electric motors (four) for the actuators and indexers (controllers) that interface to an IBM PC AT computer were specified and purchased. However, work on the vehicle was halted prior to formalizing the installation designs. The steering actuator concept was based on a 1300 oz-in. torque motor, a cog belt, and a pulley system mounted on the steering gear shaft. The brake actuator concept required a 700 oz-in. torque motor and a rack/pinion assembly to activate the regular brake cylinder. A 140 oz-in. torque motor, drum, and cable assembly would be used to control the throttle linkage. The shift lever actuator concept included a 140 oz-in. torque motor and a linkage mounted in a floor-shift box.

Vehicle status sensors for fuel level, oil pressure, charging current, engine temperature, and engine RPM were also specified and purchased.

The vehicle, unmodified, and all the motors, indexers, and status sensors were delivered to the sponsor.

LIST OF REFERENCES

1 MacAdam, Charles C., "Application of an Optimal Preview Control for Simulation of Closed-Loop Automobile Driving," IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-11, No. 6 (June 1981).

2 Zuk, David M., et al., "3D Sensor for Adaptive Suspension Vehicle," Report 164700-4F, ERIM (November 1984).

3 "Range and Reflectance Processing Workshop," ERIM Staff Report 179100-27-X, ERIM (December 1987).

APPENDIX A

AUTONOMOUS LAND VEHICLE SIMULATION

AUTONOMOUS LAND VEHICLE SIMULATION

The purpose of this report is to fully describe the functioning of the final version of the Autonomous Land Vehicle Simulation and to explain the goals that were meant to be accomplished with this software.

1.0 BACKGROUND

The last working version of the ALV Simulation is different in nature from its predecessors in that it is a synthesis of the two modes of the simulation that existed early on in development. The first mode was one in which the user planned a path that the vehicle would follow no matter if it left the road or ran directly through obstacles. The second mode used our image processing algorithms. The vehicle was given a starting coordinate, presumably on a road, and attempted to move down the road while avoiding obstacles and remaining on the road surface. The final version of the simulation used a planned path similar to the first mode; however, the final product only uses this path as a guide--an attempt is made to stay close to the path, but divergence is possible. If an obstacle is detected the vehicle will leave the planned path and, using image processing, will determine a path around the obstacle and later reconverge on the planned path.

The ALV Simulation was designed with the following purposes in mind:

1. It should be easily adaptable and expandable for the addition of new processes and experimental parameters.

2. The rudimentary control system which is sub-optimal even at low speeds should enable the collection of sufficient feedback information for researchers to determine a robust dynamic control scheme.

3. It will provide data for the study of the effects of the INS related inputs to the ability of the software to control the vehicle.

The ALV Simulation allows the examination of the reaction of the image processing when the vehicle hits potholes and makes sharp turns as well as the nature of the feedback loop between the computer commands and the subsequently produced images.

2.0 GENERAL DESCRIPTION

In the simulation, an initial path is specified by the user. It is designed so that the vehicle should stay as close as possible to the planned path without running into obstacles or going through large potholes and without leaving the road surface. Image processing algorithms are used to determine whether the vehicle must diverge from the path to avoid an obstacle and to control the manner in which the vehicle will return to the planned path. These algorithms always assume that the vehicle should continue down the road that it is on and will make a turn only if the road turns or if the planned path turns onto another road.

The ALV Simulation software is a discrete real time simulation in which the state of the vehicle is updated 160 times a second. Image inputs are given to the image processing system two times a second and the image processor outputs a midpath specification that is combined smoothly with the pre-planned path. Thus the path that is actually followed is constructed using the pre-planned path as a guide while taking into account any need to avoid obstacles or potholes. The vehicle control gives steering and speed commands required to keep the vehicle close to the constructed path. The vehicle controller works in increments of 1/160th of a second, but the vehicle responds slower and new path commands are received only twice a second.

The simulated vehicle responds as a function of its state and the commands. The vehicle state includes position, heading, pitch, and roll, and determines how the current scan line is formed. The simulated vehicle resides in a simulated world which is an array of elevations. The resolution of this world is a compromise between processing time constraints and the need for 10-20 seconds worth of data. Ray trace algorithms are used to form an image given the vehicle state and the depression angle of the scanline. The vehicle state along with error is given to the image processing system to correct for vehicle motion and heading.

The formal outputs of the simulation are specifications on the precision of the INS inputs to the image processing components. Useful outputs include the distances between the commanded path and the path the vehicle followed and feedback between vehicle control and subsequent images.

3.0 SIMULATED VEHICLE

The vehicle model consists of two parts. The first part is based on a computer program from TACOM written by F. Hoogterp. This program accepts inputs which describe the steering, acceleration, and the change of height the tire encounters on the terrain and outputs the roll and pitch. The second part of the model explains how the steering angle and speed respond to commands and how heading is affected by steering angle.

The TACOM program was a stand-alone program which read the steering, acceleration, and tire elevation profiles and generated values for pitch and roll. In the VAX VMS environment, vehicle state information was lost between calls; consequently, the program was studied and modified so that it could be called 160 times a second, with current acceleration and tire path information as input and pitch and roll information as output.

The second part of the vehicle model defined the vehicle's relationship to the world and to the control system. The vehicle control system gives steering angle and velocity commands. The vehicle responds to these commands based on its present state. If the vehicle is already undergoing an acceleration, but with the wrong sign with respect to the most recent velocity command, then the acceleration is first changed to zero and from zero it is varied linearly to the peak acceleration with the right sign. Thus, at any time, t, in units of 1/160th of a second we have the acceleration $a(t)$. The velocity at time t is denoted by $v(t)$. The velocity is always in the direction of the vehicle heading; therefore, the model cannot handle vehicle skids. The velocity is updated as follows:

$$v(t) = v(t\text{-}1) + \frac{a(t\text{-}1) + a(t)}{320}$$

The 320 in the denominator contains the delta time of 1/160 and 1/2, which averages the acceleration over the time period from $t\text{-}1$ to t.

The steering system responds to the steering command in the same fashion that the speed responds to its commands. The steering angle is denoted by $s(t)$. The change in heading as a function of the speed and steering angle depends on the distance between axles. We assume that over the 1/160 time period that both sets of wheels continue to move in their current direction (see Figure 1). Using the Law of Sines we obtain:

$$\frac{sin(h'(t))}{d} = \frac{sin(s(t))}{length}$$

where

$h'(t)$ is the change of heading of the vehicle,
d is the distance the front wheels move from time $t\text{-}1$ to t, and
$length$ is the distance between axles.

Heading, denoted as $h(t)$, is updated as follows:

$$h(t) = h(t\text{-}1) + \frac{h'(t) + h'(t\text{-}1)}{2}$$

Now that both the heading and speed of the vehicle are known, the vehicle's position can be updated. The vehicle's origin is the half-way point between the front wheels. The vehicle's position, denoted as $x(t)$, $y(t)$, is updated as follows:

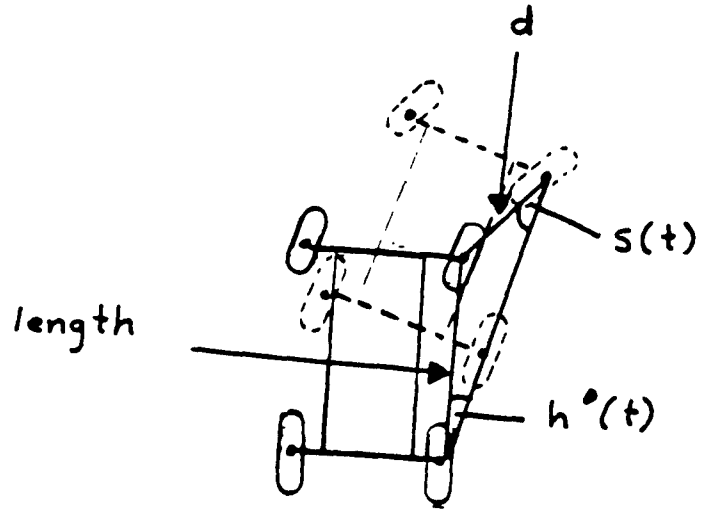


Figure 1. Derivation of Steering Angle

$$x(t) = x(t-1) + \frac{v(t-1) + v(t)}{2} * cos\frac{h(t-1) + h(t)}{2}$$

and

$$y(t) = y(t-1) + \frac{v(t-1) + v(t)}{2} * sin\frac{h(t-1) + h(t)}{2}$$

Based on the vehicle's physical characteristics and the elevation profile of the road surface at each wheel, the velocity and steering inputs result in roll and pitch angles and updated position, heading, and velocity values.

4.0 SIMULATED WORLD

The simulated world is a map with elevation values for each (x,y) point. The elevation array is a 512 x 512 image with spatial resolution varying from 3 to 9 inches and vertical resolution varying from 1 to 3 inches. The position parameters $x(t)$, $y(t)$ and $h(t)$ are with respect to this array or map. The maps can be made with any drawing system that can produce an array of integers.

The resolution parameters are inputs to the simulation, as are the boundary conditions of the vehicle's state. Figure 2 gives an example of a straight road with two obstacles. Figure 3 is a straight road with a road off to the right. Obstacles are in the same positions as Figure 2. Figure 4 gives a gently turning road with an obstacle on the right. These arrays were produced with two brownian fractals and a road center line. If the (x,y) cell was within the halfwidth of the center of the road, the elevation was taken from a low variance smooth fractal; otherwise, the elevation was taken from a second, less smooth and more varying fractal. Note that elevation differences are present on the road surface--dark is smooth, gray is medium, and bright is rough. These arrays are provided with the simulation program. The vehicle model obtains the wheel elevation from one of these maps, and the simulated sensor uses the array to obtain a range image.

5.0 SIMULATED SENSOR

The sensor's position is an inputted distance above the vehicle's origin. The sensor is assumed to be mounted on the vehicle with fixed tilt angle during any one run. The pan angle is assumed to be zero. Dynamic tilt and pan angles could be added to upgrade the vehicle controller.

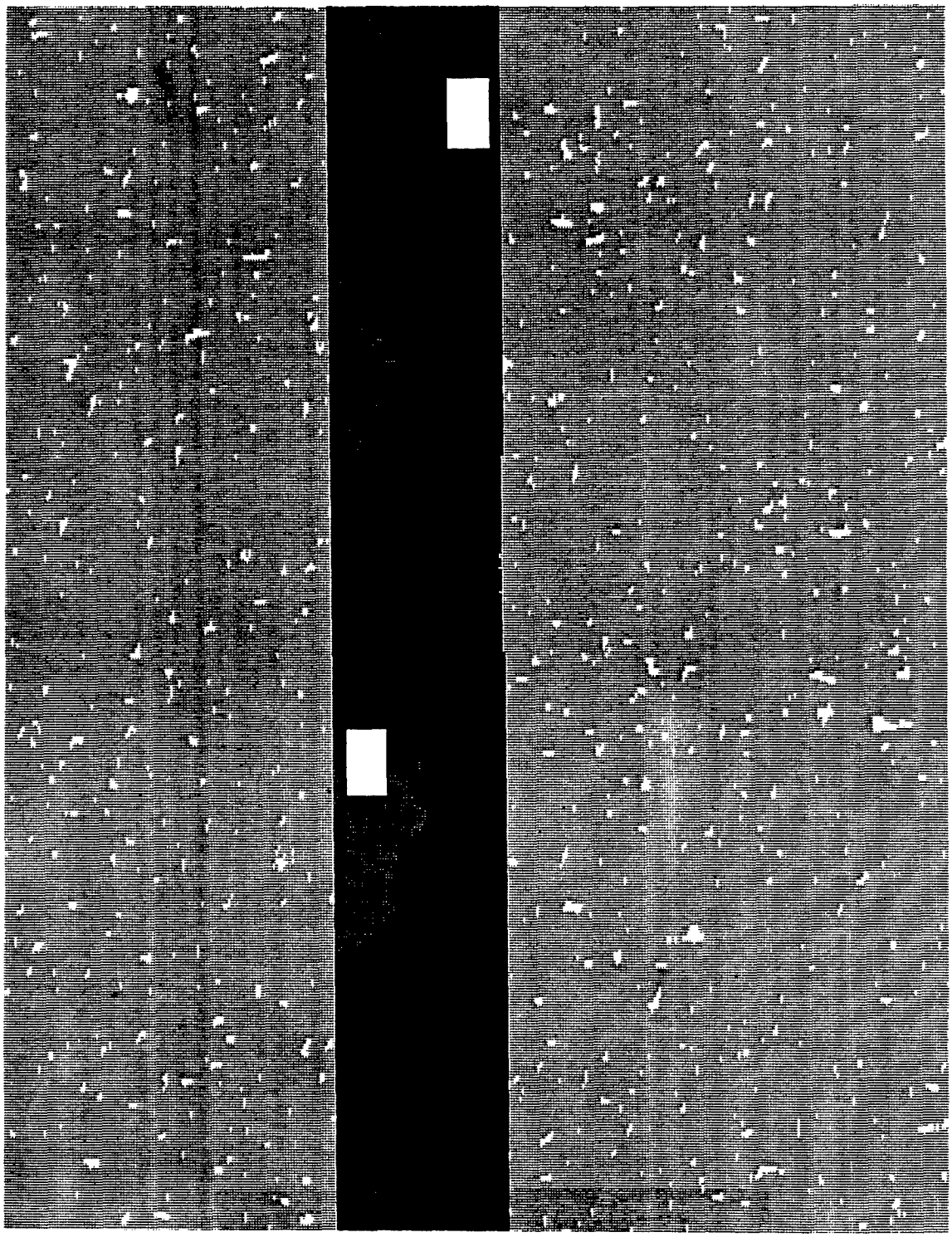


Figure 2. Map 1, Straight Road with Obstacles

Figure 3. Map 2, Straight Road with Obstacles and Intersecting Road

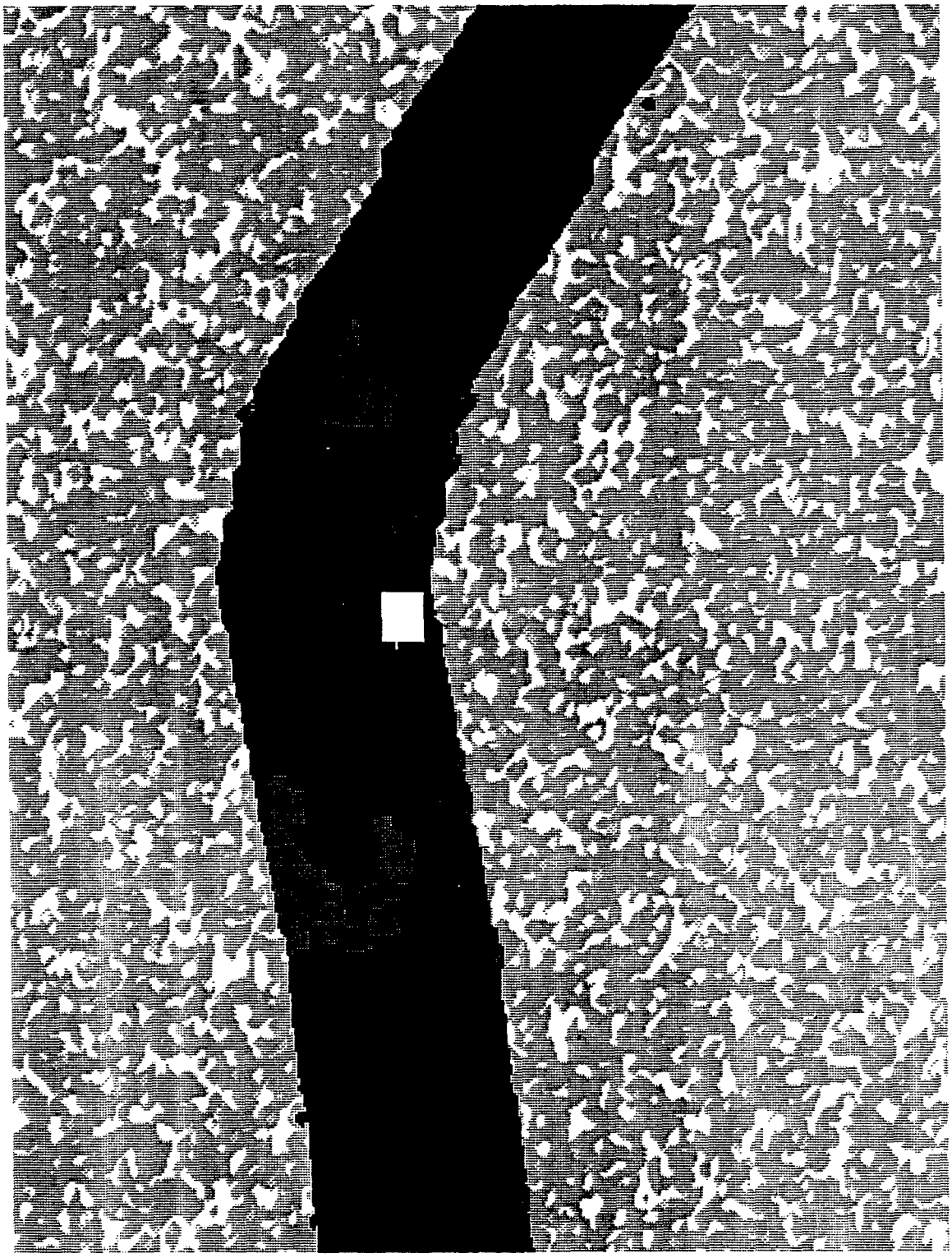


Figure 4. Map 3, Curved Road with One Obstacle

The sensor model starts with a vector with zero cross range and height and a down range component of one. The vector is rotated about the vertical axis by the azimuth viewing angle. The next two rotations are about the cross range (horizontal) axis; these rotations are the image depression angle and the tilt of the sensor. The vehicle heading, pitch, and roll rotations followed by a translation moves the vector from a vehicle coordinate system to the coordinate system of the simulated world. The model then computes the distance between the sensor and the first hit point of the simulated world if it hits; otherwise, the value reserved to flag bad data is used. A gaussian random variable is added to this distance and the sum is rounded off to the nearest integer. The *mod* 256 version of the result is saved in the image cell associated with the azimuth and depression angle. The standard deviation of the random variable is proportional to the square of the range and the area of the laser footprint. (The use of a single ray per IFOV does not allow the model to include the mixed pixel effects where a portion of the IFOV is at a significantly longer range.)

The scan pattern depression angle is modeled to change 160 times a second. Each of the two frames per second is 64 lines scanned upward and 16 lines retrace. A flag in the model can change the main scan direction to downward. The simulation was constructed to be able to accept any other scan pattern with the constraint that the range recoverer component of the image processing algorithm might have to have a different strategy. The sensor gives an image every 0.5 seconds. Associated with this frame time is the length of time from the start of frame until the controller outputs a path command. This value varies from 0.5 seconds to 2.0 seconds. This parameter allows one to study the effect of the execution time of various algorithms on the control of the vehicle.

6.0 MISSION PATH

It is assumed that the vehicle's mission is to drive from one position to another.* The paths which the simulation uses are straight lines and sections of circles inputted by the operator. The sections of circles are used to make a turn from one line to another line. The path is stored in the form:

$$\{((x,y),direction(i))\}\cup\{(v(i),len(i),radius(i),arclength(i))\}$$

where

(x,y) is the starting position,
$direction(i)$ is the initial heading of the vehicle,
$v(i)$ is the requested velocity for segment i,
$len(i)$ is the length of linear segment i,
$|radius(i)|$ is the radius of turn i,

*An ERIM-developed router based on a cost-of-traversal map may be used to find a route to the goal; the router is not included in this simulation.

if *radius(i)*>0, then left turn is commanded; otherwise
a right turn is commanded,
arclength(i) is the distance along the circle the vehicle is to drive, and
arclength(i)/|radius(i)| is the heading change that occurs with the turn.

The first group of parameters contains boundary conditions, while each succeeding group of parameters describes a straight line and a section of tangent circle. The circles are chosen to make a smooth transition from one line to the next. The length parameters can be zero if one wants to start off with a turn or to make two successive turns. The restriction on the paths help to make a timely implementation while still allowing the simulation to meet research needs. The user should examine the path and the map to see if the path is reasonable. In this version of the simulation, the vehicle will not follow the path strictly, but may deviate from the path to avoid obstacles and return after the obstacle is passed.

7.0 VEHICLE CONTROLLER

The vehicle controller has a path plan that it wants the vehicle to follow. This path is constructed from the paths obtained from the combination of the last two or three image based maps with a road midpath specification so that, if an obstacle is present, avoidance can be accomplished smoothly. There are two problems which the controller must address. The first one is to give steering and speed commands in such a way that the vehicle will approximately follow the planned path. The second problem is associated with our current method of obtaining the path from maps based on a single image rather than a set of images.

The first problem was to be approached by a method based on a model of a human driver developed by Charles MacAdam of the University of Michigan Transportation Research Institute. However, modifications of the model are necessary to account for the 0.5 second information update provided by the 3-D laser sensor. Instead of making these modifications, an alternate model was developed which works at low speed but is sub-optimal and not robust. This control algorithm uses a pursuit strategy to compute a steering command. Denote the path as p which associates a time variable t with a position $p(t)$ where t is in units of 1/160th of a second. We wish to formulate a speed and steering strategy which causes the vehicle to move approximately along the computed path from time t' to $t'+0.5$. The vehicle is assumed to be on the path at time t' which is the time that the controller receives its path command. The vehicle is not able to stay exactly on the path, but instead overshoots and undershoots about the path p. We compute the direction of $p(t+L) - p(t)$ where L is a lead time parameter chosen for a specific maneuver. The outputted steering command is proportional to the difference between the heading at time t and $p(t+L) - p(t)$. This path is then combined linearly with the preplanned path to produce a path that attempts to follow the defined path, but can avoid obstacles as well. This method is too simple to work in a rich vehicle control environment, but would allow us to meet the goals of the simulation and low speed demos.

The second problem was to be approached by using a map based on all of the recent images which viewed the area around the vehicle. This multiple image based map imposes unknown constraints on the INS system which the simulation was to address. This research program was terminated before the multiple image map feature was developed. The current image processing uses a single frame once every 0.5 seconds to extract a path. There is some disagreement between paths in successive frames. This is caused by the fact that the current image does not include the area immediately in front of the vehicle that was included in the last image. The solution is to allow the bottom 32 pixels or 8 feet of the image to be a transition zone where the planned path gracefully changes from the old path to the new one. The idea is that at the start of the transition zone the old path is best since the bottom of the current image was also seen on the last image which the path adapted to. The area above the transition zone is best represented in the new image. Denote the path from image i as $p(i,\cdot)$. The planned path is $a*p(i-1,t) + (1-a)*p(i,t)$ in the transition region. Below this region $p(i-1,\cdot)$ is used while above the region $p(i,\cdot)$ is used. The parameter "a" is chosen to be zero at the bottom of the zone and one at the top of the zone, thus effecting a graceful change from $p(i-1,\cdot)$ to $p(i,\cdot)$. If the vehicle's position is too far from the mission path, then the vehicle is stopped.

8.0 INS SIMULATION

The image processing uses only heading and distance information. The vehicle simulation provides these parameters as well as roll and pitch. All of these parameters are used in construction, but only heading and distance are used in the image processing. In fact, the image processing is not allowed to use the real heading and distance information. A gaussian random variable is added to each value and the rounding off procedure is applied. If x is the real value of the parameter, the image processing, as well as any other vehicle parameter, is allowed to see:

$$x' = \frac{[(2^b) * (s*z+x) + .5]}{2^b}$$

where

.5 is part of the rounding off procedure (integer function),
b is the number of bits used to save the value,
z is a gaussian random variable with mean 0 and variance 1, and
s is the standard deviation of the error.

Each set of possible INS components could be characterized by the parameter. The better these parameters are, the more costly they are. The output of the simulation was to assess the impact of using the various components on the quality of the maps and paths the system produced. A total cost would be computed for each configuration which included price and path quality. The configuration with the lowest cost would have been used.

9.0 SUMMARY

The ALV Simulation was limited in scope by the short development time. The simulation should allow a researcher to measure the effect of any parameter on the total vehicle control system. In the future, the simulation could be upgraded to include dynamic pan and tilt angles, a reflectance imaging capability, and the multiple image based map scheme. It must be understood that the vehicle control component of this simulation can only be regarded as temporary--it is a sub-optimal algorithm that allowed us to experiment at low speeds with the simulation. It is in no way a rich or robust method of control, and a new component should be designed using the feedback information that can be provided. Our recommendation is to measure the effects of using various INS systems for the vehicle control and then to optimize the vehicle to use both image and INS data.

APPENDIX B

ADVANCED CONCEPTS FOR AUTONOMOUS VEHICLE PLANNING

ADVANCED CONCEPTS FOR AUTONOMOUS VEHICLE PLANNING

R. Franklin and N.e. Finzel
November 1986

A Description of the Planning Problem

A plan is a partial description of an anticipated future, expressed as a schedule of events. The events are a function of the capabilities for which the planning is being done, matched to the constraints imposed by the environmental situation.

Planning must consider time and space, explicitly. Table 1 illustrates a categorization of machine planning in which spatial planning has been classified with respect to the stability of the information over time. Stable knowledge is that which does not or has not changed with time. Incremental knowledge represents those aspects of the domain of operation which do show change. Local knowledge is that information which is within the limits of available sensors while global knowledge represents information of the entire operational context. This latter knowledge is usually considered to be knowledge given a priori to the planning system at the onset of a mission.[1]

Tactical plans deal with spatially and temporally immediate situations faced by a vehicle planning system. An example of this is the choice of which fork to take in a bifurcated road. The road is a relatively stable feature in the operating domain and selection of one branch over another requires immediate attention as a vehicle approaches. Tactical planning also keeps a vehicle on course relative to some more global plan.

Strategic planning considers stable information at a global level. Such planning might consider a choice of road types -- dirt vs. paved -- for attaining a particular goal. Weather information would be an important factor in this choice, as well as fuel cost and/or cultural features such as dwellings. The list could be quite long and, as of yet, no general agreement has been reached as to a common set of features that need to be considered at a strategic level. This set will likely be some function of vehicle type and mission requirements.

Reactive planning deals with unanticipated events within the range of sensing. Detecting and avoiding obstacles in a road fall under the responsibility of this planner. In general, this level consists of contingency maneuvers for situations where a short-term deviation from the normal purposive or goal-seeking behavior of a vehicle is required.

Integrative planning occurs under conditions where the domain of operation has changed radically relative to the indication based on prior knowledge. Two examples serve to illustrate this type of planning: One occurs when the landscape has been significantly altered, as it might after a military action. Autonomous vehicles operating under these conditions may have a terrain database of the area taken by satellite six months previously and would therefore find it difficult to match incoming sensor information with prior knowledge. A second situation can arise when a vehicle becomes lost. Gross loss of knowledge concerning spatial position, as opposed to relatively minor uncertainty of position due to the vehicle sliding on the terrain surface (for example), can arise from a number of circumstances such as: failure of some equipment on board, inaccuracies of the global information resulting in a 'wrong turn' being taken, or operation under unanticipated, adverse weather conditions. In all of these cases there is a loss of correspondence between the vehicle's knowledge base and what the sensors indicate regarding the operating environment. Under these conditions the vehicle's uncertainty

---

1. This classification after a scheme presented by Dr. Ted Linden, of ADS Corp., at the DARPA Workshop on Planning, given in May 1986 in Washington, D.C.

A CLASSIFICATION OF PLANNING ACTIVITIES

	Local Knowledge	Global Knowledge
Stable Knowledge	Tactical Planning	Strategic Planning
Incremental Knowledge	Reactive Planning	Integrative Planning

Table 1: A Classification of Planning Tasks According to the Frame Of Knowledge and Its Stability Over Time.

is high and the predictive benefits of planning are significantly reduced [3]. A particular set of behaviors or type of functionality is required in this instance.

Each of these types of planning may be required at varying times with respect to one another and yet may also require predictable, periodic re-computation. Of the four, computation of those dealing with stable knowledge is most likely to be predictable. In addition, different types deal with domain information at different granularities or spatial resolution. In fact, it is most desirable to plan along a continuum of resolution of domain information, addressing different questions to different points along the continuum. This indicates a need for some structure and control of knowledge used for planning and for the rules applied to it.

The requirements of an effective planning system for autonomous mobility are extensive and computationally demanding. Such a planner should:

- have a set of survival primitives, or contingency plans, which can be put rapidly into effect.
- plan as well as possible with knowledge currently available, whatever the state of that knowledge is.
- generate and carry out plans which will augment the existing knowledge, thus enabling the vehicle to carry out a mission which would otherwise be thwarted for lack of accurate information.
- recover from loss of egocentric (vehicle based coordinates) to allocentric (world based coordinates) mapping.

A Basic Planning System

In the design of a planning system for road following, several technological advances were key to shaping the mechanisms and overall structure of the system which is being used in the TACOM/ERIM research effort. Since the ERIM scanner provides 3-dimensional information of a scene and since extensive maps are now routinely available of most areas of the world, a decision was made to represent knowledge for vehicle planning as icons or maps rather than as more highly derived, symbolic information. Given this representation, several mechanisms were required:

- Some mechanism was required which would find routes on iconic representations of the operating domain (including maps).
- A mechanism was also required which would store and maintain maps at various levels of resolution for planning purposes.

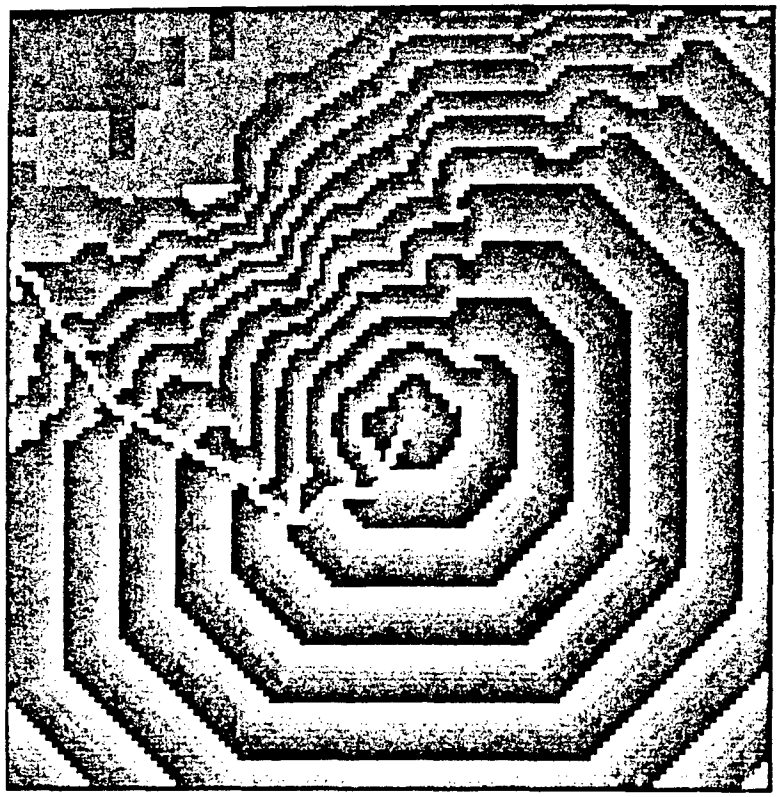


Figure 1: Result of running the routing engine on a map (128 x 128 terrain grid)

The ERIM Cytocomputer (Cyto) will process information in image (or map) format very rapidly. A program was written for the Cyto such that, given a map in which each element (or pixel) of a map was given a cost of traversal, the program would return a least cost path. This was accomplished using a wave propagation technique in which a wave front was broadcast radially from a point or origin of the map, representing the system or vehicle's current position, and spreading until the wave front covered a goal, also marked on the map [2]. This system is described more fully in Quek, et al [3].

Although the Cyto operates on an entire map, its computation speed is high, making such a 'grid-search' for a path effective in real-time. Figure 1 illustrates a route located with the use of such a routing engine. Using a 128 x 128 grid as a measure of performance, the current generation of Cyto hardware can trace a route in approximately 1/40th of a second [4].

Using a terrain generated with the use of fractal mathematics [5],[6] to provide a map containing high spatial complexity, a system was developed to maintain a stack of maps at several levels of spatial extent [3]. Figure 2 illustrates such a stack, three levels deep. In the computer simulation for which this technique was developed, the planning system was given a map of the domain or fractal world in which it was to operate, reduced by pixel averaging 16:1 (shown as MAP2 in Figure 2). The simulation system then created two maps at higher levels of resolution (MAP1, MAP0 of Figure 2) by expanding MAP2 4:1 to give MAP1,then applying a 4:1 expansion of MAP1 to create MAP0. Thus, MAP0 was equal in resolution to the level of resolution of the sensor and hence, in this particular simulation, to the resolution of the fractal world from which it was originally drawn.

These maps are used by a planning system to generate routes most likely to be least costly. They are egocentrically maintained. By that it is meant that the vehicle's position in the world is at the center of each of these maps, within a certain movement latency window. As the vehicle moves through the world, it will cross the boundary of one of these windows. That crossing triggers a scrolling action in which information from the map at a level of resolution below that of the scrolling map, and in the direction of travel of the vehicle, is expanded and added to the edge of the scrolling map. Likewise, information at the edge of the map directly

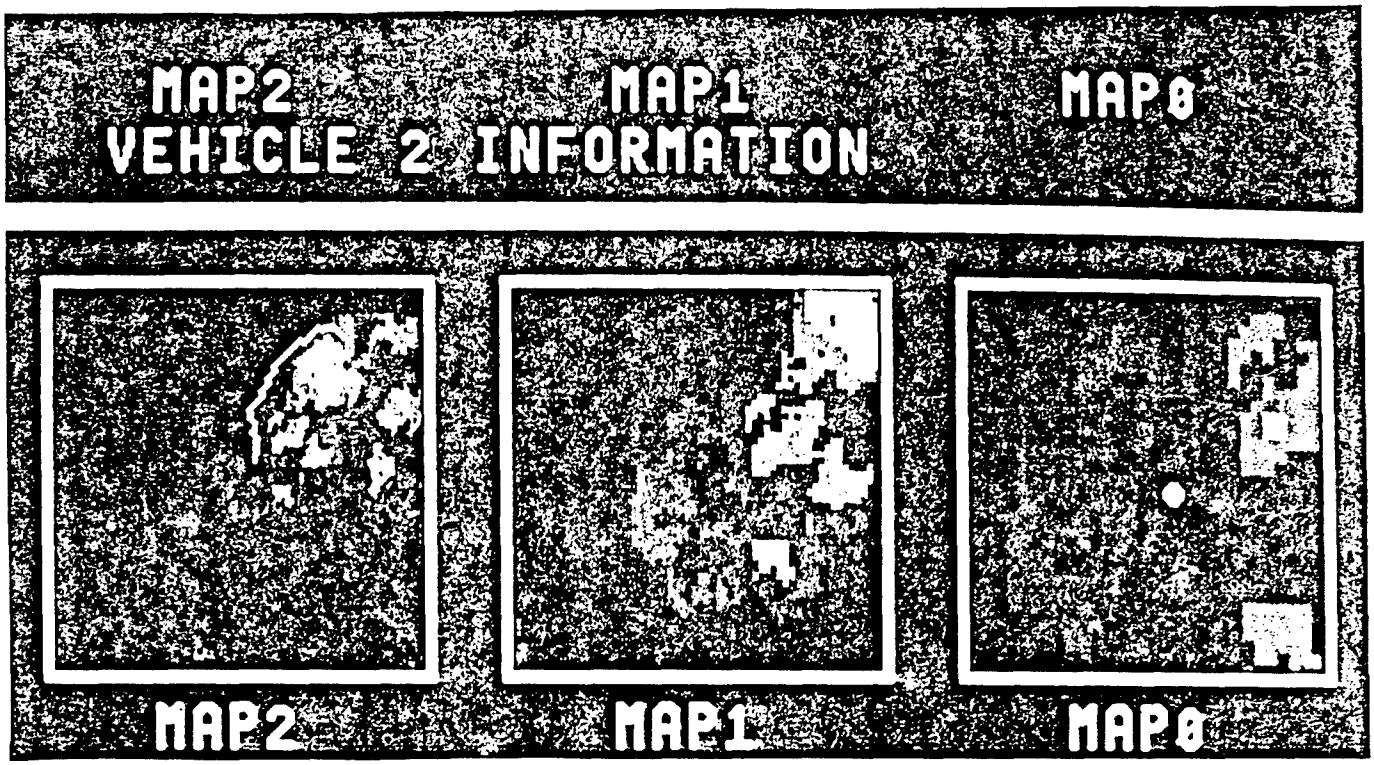


Figure 2: Maps of a Simulated Complex Terrain. MAP2 is a 16:1 down-sample of a simulation world. MAP1 and MAP0 are re-expansions of MAP2, each at a 4:1 increase. Note that no new information is created by this expansion, rather, space is filled with redundant information.

away from the direction of travel is compressed and added to the map a level below. In the current system, expansion is accomplished with a 4:1 increase in pixels, creating redundant information from the map below. Compression is accomplished in the same way, but in reverse. These actions are illustrated in Figure 3. A more complete description is available in Quek, et al [3].

The net result of this mechanism is a tool for managing prior map information and incorporating incoming sensor information into this map database. A routing engine can then be operated at any level of map to find a lowest cost path. Depending on the scope of the various maps in the stack, route planning can be done at any level from most global to most local.

Organization of the TACOM Vehicle Planner

The planning system proposed for the TACOM vehicle embodies an overall flow of information from raw images captured by vehicle sensors to short and long range motion decisions produced by the planner. Images captured by the ERIM scanner will be processed to correct for the effects of vehicle motion. These compensated images will be entered into a queue of images, whose depth is a function of vehicle velocity. This queue will serve as raw material for any number of perception processing tasks called virtual sensors. Virtual sensors are methods for extracting particular kinds of information from different numbers of images. For example, a virtual sensor to find road edges may need to process every incoming frame. But a virtual sensor which detects large vertical objects in the road may need to look at the last three frames and may need to do so only on every fourth frame.

The idea of decoupling the planning system from the direct sensor input -- that is the idea of virtual sensors -- was developed in response to the perceived need for rapid and flexible processing for autonomous vehicle control [7]. Virtual sensors and a queue manager, or more correctly a form of blackboard system, allow the planner the flexibility to request perceptions of

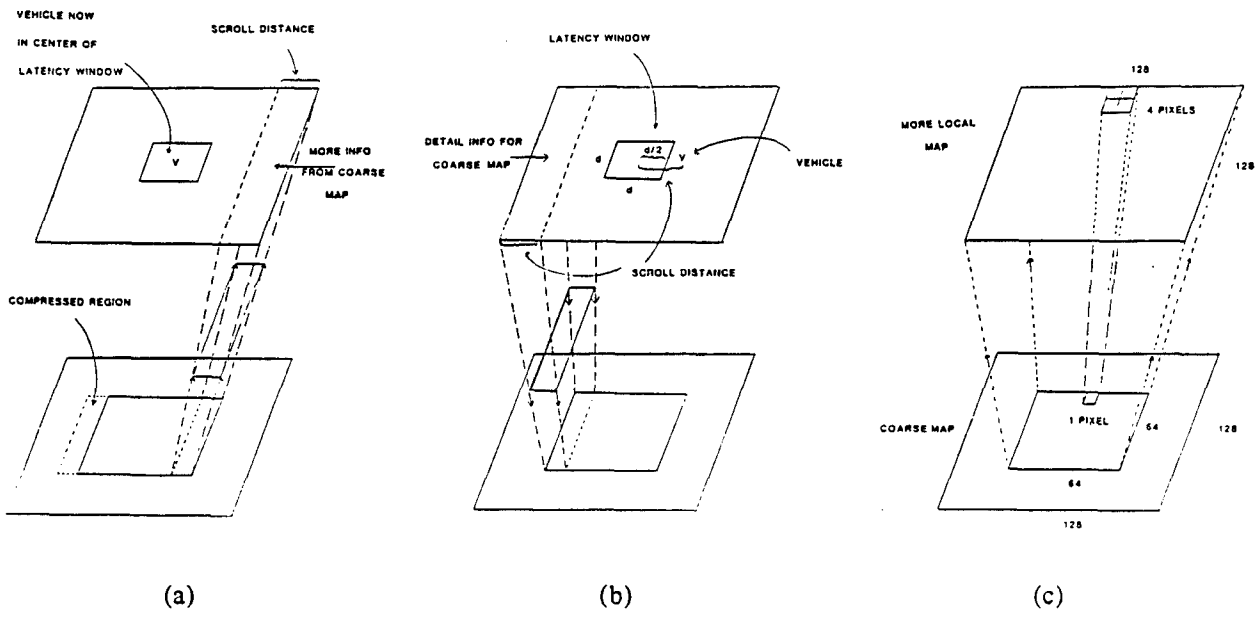


Figure 3: The mechanism for scrolling maps and expanding and compressing information to and from successive map levels. (a) Shows expansion of coarse map information into detailed map. (b) Shows scrolling triggered by vehicle crossing between windows and compression of map information. (c) Shows creation of a redundant map of higher resolution.

the data that are consistent with evolving conditions in the operating environment and simultaneously preserve the potential for concurrent processing.

Blackboard Architectures for Planning

The blackboard architecture was first developed as a problem solving tool for the HEARSAY-II speech understanding system [11]. Its purpose was to reduce the number of diverse interpretations of spoken sounds into a coherent, syntactically correct expression, i.e., to translate sound waveforms into a meaningful sequence of words. This problem is a specific example of a more general area of problem solving characterized by a need for diverse kinds of knowledge for its solution and by large amounts of uncertainty and variability in its input data. The blackboard architecture has proven useful in coping with the representation and control issues inherent in this type of problem.

A blackboard architecture consists of three basic components: knowledge sources, a blackboard, and a control structure. Knowledge sources (KSs) are diverse, independent programs whose function is to generate, combine, and evaluate possible solution elements (hypotheses) to the given problem. Each KS has a condition-action format, where "the condition component prescribes the situations in which the KS may contribute to the problem solving activity, and the action component specifies what that contribution is and how to integrate it into the current situation."[11] The situation referred to in this description is the set of existing data about the problem, both from the problem statements and from hypotheses previously generated by KSs. Therefore, the hypotheses generated by KSs serve in two ways: they are potential solution elements to the given problem, and they are used as data by the condition components of other KSs.

Since each KS is an independent module, communication between KSs must occur through some separate medium. This medium is the blackboard: a structured, global database which records KS generated hypotheses. The blackboard is subdivided into information levels corresponding to intermediate representation levels of the problem solving process. Each level

interacts with a specific set of KSs which generates hypotheses of appropriate detail for that level. In this manner the blackboard and KSs share a hierarchical structure such that data contained and generated at each level are abstractions of data at lower levels.

The blackboard/KS interactions are regulated by a control structure which consists of a loop containing three distinct steps, as follows: First, the conditions for activation of each KS are compared against the data existing on the blackboard. Those KSs whose conditions are satisfied are grouped into a set of eligible KSs. Second, control rules are used to choose one KS from the eligible set. Finally, the chosen KS is activated, which generates or modifies blackboard data. This cycle continues until the blackboard indicates the completion of a solution.

Recently, the concept of a blackboard *control* architecture was introduced by Hayes-Roth[12]. This concept extends the basic blackboard architecture by explicitly dividing a problem solving process into two separate areas: domain and control. In this new architecture, the domain blackboard holds information specific to the problem being solved. Another blackboard, a control blackboard, contains information about solving problems, i.e., 'which of its potential actions should the AI system perform at each point in the problem-solving process'[12]. Two distinct sets of knowledge sources exist, corresponding to the two blackboards. The domain KSs are domain specific, i.e., they relate to the problem being solved and act primarily on the domain blackboard. The set of control KSs operates primarily on the control blackboard, but consists of both domain specific and domain independent KSs.

Both blackboards are controlled by one loop containing the three control steps given above. Each step is accomplished via a 'basic' control KS. Because these three control KSs schedule both domain and control KSs, they indirectly modify their own behavior based on circumstances occurring in the problem solving process. In this way, the blackboard control architecture adapts its methods of solution to the specific problem it is solving.

A Blackboard Architecture as a Planner

A blackboard control architecture as given by Hayes-Roth is currently being investigated by Chappell[10], of the University of New Hampshire. This work concerns the implementation of a prototype path planner for an autonomous submersible survey vehicle. The conceptualized planner is to be built into a blackboard control architecture.

Adhering closely to Hayes-Roth's definitions, Chappell first divides the planning problem into domain and control areas. Each area is hierarchically subdivided into abstraction levels, based on the time required for a typical task in the level to execute. In this case the lowest tasks in the hierarchy are those tasks required to operate in under a second, and the highest are those allotted from ten seconds to over a minute for completion. The levels of abstraction for the domain blackboard are defined as **Mission**, **Design**, **Procedure**, and **Segment**. **Mission** specifies tasks to be included in the plan; **Design** specifies the spatial/temporal layout to follow while planning; **Procedure** sequences individual path segments into partial paths; **Segment** generates path segments to connect partial plans and details the performance envelopes for those plans.

The domain independent control architecture is divided into five abstraction levels: **Problem** - chooses one of the existing problems to solve, **Strategy** - selects a procedure for solving the problem, **Tactic** - the general method to be used while generating a plan on the domain blackboard, **Focus** - the abstraction level of action on the domain blackboard, and **Policy** - the scheduling criteria to use when choosing knowledge sources for execution. With this blackboard design, an integrated domain and control solution is expected, as shown in Figure 4, for a task of surveying two rectangular areas. The details of the individual abstractions levels have not been developed, but the paper does provide a conceptual implementation of a blackboard control architecture.

A Blackboard Architecture as a Planner Component

The use of the blackboard architecture as a component of a planner for an autonomous land vehicle (ALV) has been explored by Payton[7]. This planner does not include a blackboard

CYCLE	KNOWLEDGE SOURCE	DECISION	BLACK-BOARD	LEVEL
1	accept_problem	solve_problem	control	Problem
2	set_policy	favor_recent_Ks's	control	Policy
3	set_strategy	plan_a_mission	control	Strategy
4	implement_strategy	successive_refinement	control	Tactic
5	implement_tactic	domain_level=Mission	control	Focus
6	set_mission	include(s, A1, A2, f)	domain	Mission
7	set_start	start=(x1, y1)	domain	Procedure
8	set_finish	finish=(x2, y2)	domain	Procedure
9	change_focus	domain_level=Design	control	Focus
10	locate_task	create_survey2	domain	Design
11	locate_task	create_survey1	domain	Design
12	change_focus	domain_level=Procedure	control	Focus
13	gen_search_path	segment_list2	domain	Procedure
14	gen_search_path	segment_list1	domain	Procedure
15	change_focus	domain_level=Segment	control	Focus
16	expand_segment	finished_path	domain	Segment
17	expand_segment	finished_path	domain	Segment
18	expand_segment	finished_path	domain	Segment
19	expand_segment	finished_path	domain	Segment
20	dectect_plan	execute_the_plan	control	Strategy
21	send_plan			

Figure 4: Decision Log for Proposed Blackboard Control Architecture. (Chappell, S., "A Prototype Trajectory Planning System for an Autonomous Vehicle", Proceedings Blackboard Workshop, Carnegie-Mellon University, June 12-13, 1986)

control architecture, but does use a modified *domain* blackboard. Since the paper addresses ALVs as opposed to submersibles, the concepts presented are more directly applicable to our work.

This planning system involves a hierarchy of control levels, "in which lower level modules perform tasks requiring greatest immediacy, while higher level modules perform tasks involving greater assimilation of sensor data"[7]. Immediacy refers to the fact that the faster any sensory data can be used to effect action, the more value they have for control (and the less obsolete they are due to vehicle motion). Assimilation includes both extensive sensor image processing, and the merging of various sensor modality outputs into a single result. Though time consuming, assimilation provides more complete and detailed information about terrain features to the planner. The trade-off between immediacy and assimilation is therefore one between processing time and data completeness. The levels of the control hierarchy are intended to run in parallel, covering the entire spectrum of immediacy/assimilation trade-offs[7].

The hierarchy presented by Payton ranges from the level of greatest assimilation - **Mission Planning** - through **Map Based Planning** and **Local Planning** to the level of greatest immediacy - **Reflexive Planning**. Each level of the system receives input from a comparably layered perception system. The **Mission Planning** level translates mission goals into geographic goals and mobility constraints. These are translated into specific route plans by the **Map Based Planner** using map-based reasoning and long term assimilated data from perception. The **Local Planning** module insures that a route plan gets properly executed by selecting reflexive actions appropriate to that execution, and the **Reflexive Planning** module maintains real-time vehicle control.

Payton describes the level of reflexive planning in greatest detail, as he considers it to be the base level of competence which will remain intact as higher levels are added. The **Reflexive Planning** module is defined as a large collection of expert sub-modules, each divided into two distinct elements, a perceptual component called a 'virtual sensor', and an action component called a 'reflexive behavior'. Virtual sensors are defined as "black box sensing

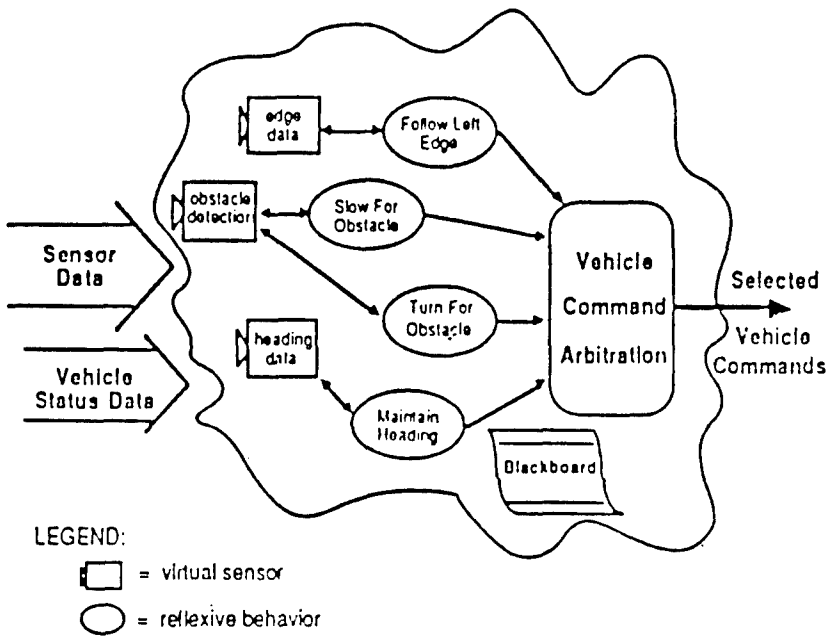


Figure 5: A set of reflexive behaviors and their virtual sensor operating within the reflexive planning module. (Payton, D., "An Architecture for Reflexive Autonomous Vehicle Control", Proceedings 1986 International Conference on Robotics and Automation)

devices which can detect very specialized environmental features. Reflexive behaviors are highly procedural units due to the need for immediacy."[7] Under normal circumstances, several virtual sensors and reflexive behaviors are operating asynchronously and in parallel. Reflexive behaviors write their vehicle control decisions to a common blackboard, where a command arbitration unit selects the highest priority commands and issues them to the vehicle actuators. Figure 5 illustrates this behavior.

The area of interest at the reflexive level is the blackboard link between the reflexive behaviors. Blackboard entries contain a command and a value. Any active behavior may issue a command to the 'speed' or 'turn-rate' procedures via the blackboard. Command arbitration rules determine the acceptance of each command. Commands of higher priority supersede those of lower priority. Priorities of the reflexive behaviors themselves affect arbitration if two commands with the same priority are issued.

Comparing Payton's use of a blackboard with the definition, an attempt was made to identify the three basic components of a blackboard architecture:

- Knowledge Sources — Strictly, there are none. Neither the virtual sensors nor the reflexive behaviors are invoked by a control structure. Both, however, write results to the blackboard which have a bearing on the vehicle action taken. Virtual sensors operate continuously, writing sensor data to the blackboard. Reflexive behaviors process the most recent results of a predetermined set of virtual sensors, and issue commands. These commands are placed on the blackboard for 'activation' by the control structure.

- Blackboard — The blackboard is the area to which virtual sensor results and commands generated by the reflexive behaviors are written.

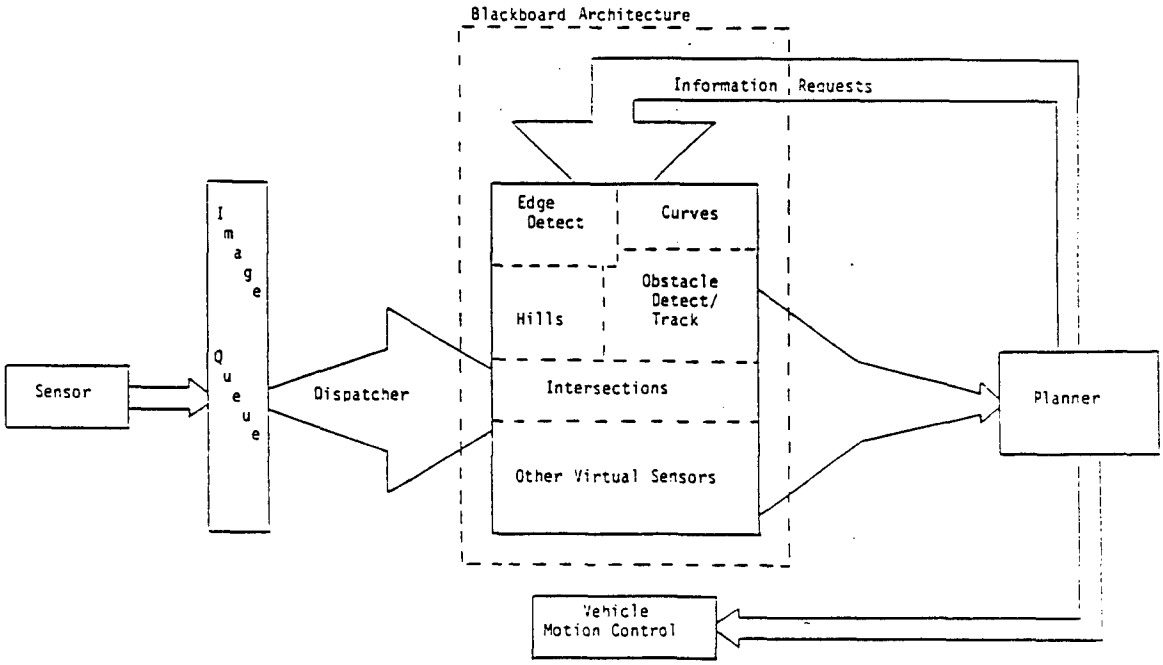


Figure 6: Overview of Planning System Showing the Role of the Blackboard Architecture.

- Control Structure — The control structure is the command arbitration unit. This unit chooses and activates commands based on their relative priorities. While this is one possible function of a control structure, it does not strictly fit the blackboard architecture definition because no control of knowledge sources is entailed.

Payton has implemented a blackboard without the strict blackboard architecture. However, his approach works. The reflexive level has been implemented in software. A planar approximation of the real world was used for terrain. The simulated vehicle was rectangularly shaped, included a compass, an odometer, and a simplified ray-trace range scanner, and was controlled by speed and rate of turn commands. According to Payton, this simulation "demonstrated meaningful levels of performance competence". Work on the **Local Planner** and on reflexive behaviors applicable to an ALV continues.

A Proposed Blackboard Implementation

The blackboard architecture we propose for implementation on the ERIM/TACOM vehicle is significantly different from both examples given above. We are differentiating between a blackboard and a planning system. Our blackboard architecture is intended to service information requests from, and return results to, a separate planner. Figure 6 outlines the proposed system. As diagramed, the blackboard is to act as a common message handler for the virtual sensors, image dispatcher, and the planner. As defined, a blackboard architecture contains three basic components, namely, knowledge sources, a blackboard, and a control structure. We propose the following descriptions for these components.

Knowledge Sources

We intend to implement Payton's concept of a virtual sensor as a kind of knowledge source. Though not the sole type of KS that we will employ, virtual sensors will play an important role in the architecture. The primary condition for activation of this type of KS will

be a request from the planner for specific, image derived information. Such requests will be queued for activation. Activation of a virtual sensor will result in specific terrain information on the blackboard, for access by the planner as required. At present, virtual sensors will be incorporated as lower level KSs. Higher level KSs may be used to merge virtual sensor results to provide a more global representation of the world to the planner. An example of this is the recognition of an intersection. Virtual sensors may place results on the blackboard indicating a road to the right , a road to the left, and a road sign. A higher level KS would recognize these world features as indicative of an intersection and place this information on a higher level of the blackboard.

In summary, lower level KSs will be implemented by virtual sensors whose condition-action format is 'if requested, process X images and return an indication of the presence and/or location of a specific feature'. Higher level KSs will assimilate virtual sensor results. The condition action format of these KSs will be 'if the elements indicating a specific terrain condition have been found, place a message on the blackboard indicating that condition'.

Blackboard

The blackboard will be a global database, as in the standard blackboard architecture. Information on the blackboard can be thought of as objects belonging to one of at least three classes. These classes are (1) Information Request, (2) Image Request, and (3) Result.

The Information Request class in intended to contain messages from the planning system. These messages will consist of a label indicating the information wanted, a value indicating priority, and an identification number. The message may also contain a status, indicating whether or not the request has been serviced. These messages will remain on the blackboard until the request has been service or canceled.

The Image Request class will contain messages from the virtual sensors. An active virtual sensor may place a message on the blackboard requesting sensor images. The message may contain information about the total number of images requested, the start image of the sequence, and the priority with which the images are needed. The priority will most likely be that of the virtual sensor making the request. These messages will also remain on the blackboard until the requests are either serviced or canceled.

The Result class retains messages from the virtual sensors and other knowledge sources. These messages will contain the results of a given task identification. The task label and identification may be bound to the result. Both the planning system and the higher level knowledge sources will need to access results, so locating a task's result quickly is imperative. It is possible that the results may be deleted from the blackboard after enough time has passed to make them obsolete.

Summarizing, the blackboard will contain all information requests from the planning system, all image requests from the virtual sensors, and all results from the virtual sensors and the higher level knowledge sources. The information will remain on the blackboard until explicitly deleted.

Control Structure

The mechanism for the control of the blackboard has not yet been determined, but we know some of the areas which the control structure must handle. An important role of the control structure is that of comparing KS activation conditions against the current data on the blackboard. The control structure will have to check the KS invocation conditions, mark those KSs whose conditions are satisfied, and invoke one of the marked KSs based on priority. Control is responsible for ensuring that the lower priority information requests are not lost. It may also need to upgrade the priority of tasks whose results are re-requested or upon reception of an urgent message form the planning system.

In summary, the blackboard architecture is to:

- Hold planner requests and priorities

- Check conditions for virtual sensor invocation
- Invoke virtual sensors whose conditions are met
- Hold virtual sensor image requests and priorities
- Hold virtual sensor results

Note that the above description of the proposed blackboard architecture differs from those in the literature. Chappell uses a blackboard control architecture to solve the whole problem; he has no separate planner. The design of a blackboard control architecture is followed closely, using all the components: domain blackboard and knowledge sources, control blackboard and knowledge sources, and one control loop.

Payton uses a modified blackboard architecture (BBA) to hold several different solutions so that a choice can be made between them. He does not use a standard BBA design. Strictly speaking, there are no knowledge sources and the control structure is used in a limited way. The BBA does not do any planning; there is a separate planning system.

We plan to use a blackboard architecture to interface a planning system with a sensor or sensors. So our BBA will not be doing the planning but giving the planner information it needs in a form it can use most efficiently. We intend to follow the standard design of a BBA, using knowledge sources, a blackboard, and a control loop. Only our use of it is different; it will not solve a problem, it will assist in solving a problem.

BIBLIOGRAPHY

1. Chattergy, R., 1985, "Some Heuristics for the Navigation of a Robot", Int'l J. Robotics Res. 4(1), 59-66.

2. For a discussion of this technique, see: Diehl, R.K., McGhee, R.B., Zyda, M.J., 1986, "Two-dimensional Polygonal Representation of Maps for use with Autonomous Vehicle Route Planning", Rep. #NPS52-86-016, Dept. of Computer Science, Naval Postgraduate School, Monterey, CA.

3. Quek, F.K.H., Franklin, R., Pont, W.F., 1985, "A Decision System for Autonomous Robot Navigation over Rough Terrain", Proc. SPIE Conf. on Intelligent Robots and Computer Vision, V.579 #59-50, p 377-388, Boston.

4. Proceedings of the DARPA Workshop on Processing 3-Dimensional Range and Reflectance Information, Franklin, R. & Eberle, K. (eds.), Ann Arbor, MI, from ERIM, P.O. Box 8616, Ann Arbor, MI 48107.

5. Pont, W.F., 1983, "Fractal Background Models in Scene Simulation", IR&D Report, Project #659111, ERIM, Ann Arbor, MI.

6. Gaddis M.E. and Zyda, M.J., 1986, "The Fractal Geometry of Nature: Its Mathematical Basis and Application to Computer Graphics", Rep. #NPS52-86-008, Dept. of Computer Science, Naval Postgraduate School, Monterey, CA.

7. Payton, D.W., 1986, "An Architecture for Reflexive Autonomous Vehicle Control", Proc. IEEE Conf. on Robotics and Automation, V.3, 1838-1845.

8. Zuk, D., Pont, W.F., Franklin, R., Dell 'Eva, M., 1985, "A System for Autonomous Land Navigation", 1985 Meeting of the IRIS Specialty Group on Active Systems, V.1, Naval Postgraduate School, Monterey, CA.

9. Blidberg, D. Richard, 1984, "Time-Ordered Architecture for Knowledge-Based Guidance of an Unmanned Untethered Submersible", IEEE Oceans, pp. 571-575.

10. Chappell, S.G., 1986, "A Prototype Trajectory Planning System for an Autonomous Vehicle", Blackboard Workshop at Carnegie-Mellon University, June 12-13.

11. Erman, Lee D., et al., 1980, "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, Computing Surveys, V.12, No.2, pp. 213-252.

12. Hayes-Roth, Barbara, 1985, "A Blackboard Architecture for Control", Artificial Intelligence 26, pp. 251-321.

13. Nitao, John J., Parodi, Alexandre M., "An Intelligent Pilot for an Autonomous Vehicle System", Proc. 1985 IEEE Second Conf. on Artificial Intelligence Applications, pp. 176-183.

14. Pearson, Glen, & Kuan, Darwin, "Mission Planning System for an Autonomous Vehicle", Proc. 1985 IEEE Second Conf. on Artificial Intelligence Applications, pp. 162-167.

15. Shafer, Steven A., Stentz, Anthony, and Thorpe, Charles E., "An Architecture for Sensor Fusion in a Mobile Robot", Proc. 1986 IEEE Int'l Conf. on Robotics and Automation, pp. 2002-2011.

16. Yang, Ju-Yuan David, Huhns, Michael N., and Stephens, Larry M., "An Architecture for Control and Communications in Distributed Artificial Intelligence Systems", IEEE Transactions on Systems, Man, and Cybernetics, May/June 1985, pp. 316-326.

APPENDIX C

BLACKBOARD ARCHITECTURE FOR PLANNING

Final Report

Nancy Finzel & Laurel Harmon

December 21, 1987

Table of Contents

1

INTRODUCTION

This report documents the design and implementation of the blackboard architecture supported by TACOM.[1] The goal of this project was to demonstrate the utility of the blackboard software architecture as a basis for flexible decision systems in planning problems. The TACOM BBA analyzed 3d (range) images for the purposes of road-following. By using previously collected sensor images, the continuous operation of a real range scanner was simulated. Each image set in motion a default data-driven processing sequence, which resulted in either identification of road edges or appropriate warnings.

Two areas were selected in which to demonstrate the potential flexibility of the BBA: 1) concurrent and independent processing of multiple images; and 2) parallel use of top-down (request-driven) and bottom-up (data-driven) reasoning. The prototype blackboard demonstrates the ability of the system to handle real sensor data, to process data concurrently (whether data-driven, request-driven, or both), to propagate requests internally and to monitor for exceptional situations. Throughout the work, an attempt was make to stick closely to the definition of a blackboard architecture as described in the literature and summarized below (Section 2).

The remainder of the report is organized into three major sections. The first (Section 2) is concerned with the definition of the blackboard architecture. In the second (Section 3), the demonstration scenario is described together with a general system design. This section provides some insight into the process of decomposing a problem into components suitable for implementation within a BBA. Section 4 presents details of our specific implementation of the BBA in terms of Symbolics Common Lisp. Technical terms from object-oriented programming are explained as appropriate.

The system is summarized in Section 5, which includes a discussion of how it compares with other known BBAs and directions for future research. The Appendix contains the entire set of code for the blackboard architecture, including its display. References in square brackets may be found in Section 6, the bibliography.

---

1. A final demonstration of the blackboard architecture was presented to the sponsors on 19 August, 1987.

2

BLACKBOARD ARCHITECTURE- A WORKING DEFINITION

Since its implementation in the 1970's for natural language understanding systems, the blackboard has been recognized as a useful software tool. It was designed for problems whose input data are highly uncertain and variable and whose solutions require diverse forms of knowledge. As a software architecture, the blackboard provides a structure for decomposing complex problems and flexibly combining subproblem solutions. The blackboard also permits multiple problem-solving strategies, such as top down (goal- or model-driven), bottom up (data-driven), and hybrid approaches, to be organized into a single system. Abstractions of the original blackboard structure have been modified for such diverse applications as interpretation of continuous passive sonar signals from multiple arrays [9], interpreting data from multiple sensors [6,7,13], as well as guiding autonomous submersible vehicles [1] and space craft [14].[2]

The same characteristics of the architecture which led to its implementation in speech-understanding systems make it a powerful tool in planning applications for autonomous land vehicles [2,4-6,10-12]. Among other things, planning involves the interpretation of sensory information, the updating of information about immediate surroundings, and the use of that information for navigation and other purposes. It entails operation with incomplete and uncertain input information and flexible responses to an unpredictable environment. The TACOM prototype blackboard architecture illustrates the use of this architecture in the planning domain.

The general blackboard architecture (BBA) specifies a framework within which to organize processing algorithms, raw data and partial results. A BBA is comprised of three basic components: knowledge sources, blackboard database, and control structure, shown schematically in Figure 1.

Knowledge Sources.

Knowledge sources (KSs) are diverse, independent programs whose functions are to generate, combine, and evaluate possible hypotheses (solution elements) to a given problem. Each KS has a condition-action format, where "the condition component prescribes the situations in which the KS may contribute to the problem solving activity, and the action component specifies what that contribution is and how to integrate it into the current situation" [3]. "Situation" in this description is the set of existing data about the problem, both from problem statements and from previously-generated hypotheses.

Blackboard Database.

---

2. See [9] for a review.

BLACKBOARD ARCHITECTURE

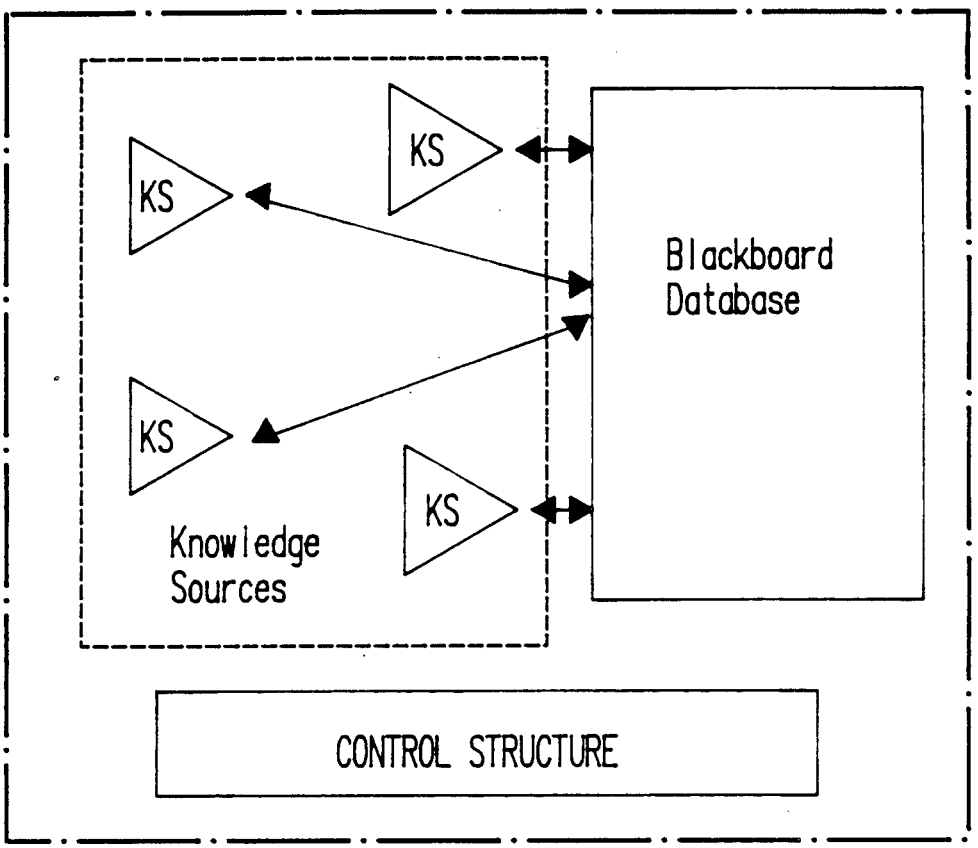


Figure 1.

Since each KS is an independent module, communication among them must occur through a separate medium. This medium is the blackboard: a structured, global database which records KS-generated hypotheses. The blackboard is subdivided into information levels corresponding to stages of problem-solving. Each level is accessed by a specific set of KSs to generate hypotheses of appropriate detail for that level. In this manner the blackboard and KSs share a hierarchical structure such that data contained and generated at each level are abstractions of data at lower levels.

Control Structure.

Blackboard/KS interactions are regulated by the control structure. The control structure has three distinct steps: First, the conditions for activation of each KS are compared against the data existing on the blackboard. Those KSs whose conditions are satisfied are grouped into a set of eligible KSs. Second, control rules are used to choose one or more KS from the eligible set. Finally, the chosen KS is activated, generating or modifying blackboard data. These steps are performed in a cycle until the blackboard indicates that a solution has been reached. Since there is no a priori restriction on the number of knowledge sources which can act simultaneously, multiple analysis paths can be pursued in parallel and abandoned as they are found to be fruitless. Uncertain and partial results arising from different processing approaches or from different sources can be combined at any point during the analysis, expediting progress to a solution.

3

DEMONSTRATION SCENARIO AND SYSTEM DESIGN

Autonomous road-following was selected as the problem domain for the blackboard demonstration. The specific sub-problem addressed is that of identifying open space for navigation, given a description of the terrain in front of the vehicle. Terrain information was derived from images collected from a 3D range sensor mounted on a vehicle as it was driven down a road. Raw images had previously been transformed to Cartesian coordinates, marked with respect to change-in-height (texture) information (the z coordinate), and stored as plan views.[3] Images were made available to the system as fast as they could be read off disk, in order to simulate continuous sensing. Given such data as input, the system task was to determine the location of road edges and clear space for road-following.

This problem was chosen because it is well-defined and limited in scope, yet retains enough uncertainty to demonstrate the potential of the blackboard architecture for planning. The problem had previously been solved in a traditional architecture, so that the necessary algorithms were available. Efforts could therefore be concentrated on the blackboard architecture development itself. Problem decomposition was straightforward, permitting incremental increases in complexity as the BBA was refined.

3.1 Problem Decomposition

In order to approach this problem within the blackboard architecture, it was broken down into subproblems with the following characteristics:

Each produces a simple, identifiable result which contributes to the overall problem solution.

Each can be formulated so that it is clear when the situation (current blackboard state) calls for its solution.

Each requires only distinct, recognizable pieces of data in order to form a hypothesis.

Subproblems interact with each other only via initial input data.

The blackboard and basic knowledge source structure development was begun around several well-defined, low-level image processing steps. Once the basic elements of the architecture were in place, features were implemented to handle the inherent

---

3. Two-dimensional arrays with indices corresponding to x and y coordinates and whose contents represent height (the z coordinate).

uncertainty of the problem, such as insufficient or missing data. Low-level subproblems meeting the criteria outlined above were determined to be:

- Find-max-row: Find the top limit of the image data array which is amenable to processing.
- Find-min-row: Find the bottom limit of the image data array which is amenable to processing.
- Find-edges: Find markings to either side of the center of the image which indicate a change of height and possibly the road edge.
- Find-left-side: Compute parameters of a best fit line for the left-edge data.
- Find-right-side: Compute parameters of a best fit line for the right-edge data.
- Fake-left-side: Infer the position of the left side from previous data.
- Fake-right-side: Infer the position of the right side from previous data.
- Find-center: Define a center-line of the road from left and right side information.

The following subproblems were identified in the problem of uncertain or missing information:

- Left-missing: Flag the inability to detect a left road side.
- Right-missing: Flag the inability to detect a right road side.
- Both-missing: Audibly indicate the inability to detect either road side.
- Found-left: Flag that the left road side has been detected.
- Found-right: Flag that the right road side has been detected.
- Check-clear: Determine if clear area exists in front of vehicle.

The solution to each of these subproblems was embedded in a separate knowledge source, as discussed in Section 4.

3.2 Blackboard

The following qualities were desired for the blackboard, or global database, and the data contained within it:

- easy placement/retrieval of data
- efficient database searches
- simple database alteration/expansion
- prevention of database read/write conflicts

- ability to store and retrieve entire database.

Several Lisp structures were considered for the blackboard: arrays, lists, property lists, association lists, tables, and records. Records and tables, specifically hash-tables, were the most promising candidates, based on the criteria above. The hash-table structure was chosen as the basis of the blackboard (see Section 4). Hash-tables are extensively supported by Symbolics software. Hashing functions exist, enabling efficient searches. Add and delete functions which avoid read/write conflicts were available. The size of hash tables changes dynamically and they can be saved to disk.

An attractive feature of records was that everything, including the database itself, is represented as a Lisp object. All data exist both in the Lisp environment and as records in disk files. Perhaps most intriguing was the concept of a "changes" file. According to Symbolics' documentation, this is set up so that systems need only process changes since the last time the database was contacted [15]. This concept appeared useful as an optimization for large blackboards. However, since low-level access functions did not appear to be implemented as fully for records as hash tables, records were not considered further.

It then remained to determine how best to store the data. We wanted the data to be easy to store, find, and retrieve. There would be several distinct kinds of data in the database: images, numbers, lists of points, results and requests. The system must be able to differentiate between data types, yet handle all data similarly. To meet these criteria, we used *objects*, defined using the Symbolics Flavor System.[4] The actual implementation of data objects is discussed in more detail in Section 4.

3.3 Knowledge Sources

The following features were considered to be essential for all knowledge sources:

- data and/or request driven operation
- generation and propagation of requests for missing data
- efficient checking for applicability
- independent operation
- propagation of priority information with data and requests

In addition, the following implementation criteria were established:

- clarity of representation
- prevention of redundant operation (duplicate KSs)
- ease of adding new KSs to system
- ease of controlling operation

---

4. See footnote 6.

- ability to apply KSs to different data simultaneously.

Several alternative KS forms have been reported in the literature.[5] These seem to differ most in the checker component of the KS. Examples are described below, with names of representative systems using each form noted in parentheses:

1. Condition/action, where condition and action are two separate procedures. When executed, the condition part searches the blackboard for hypotheses of interest to the action part; all relevant hypotheses found are passed to the action part. Upon activation, the action part processes all the hypotheses passed to it. (HEARSAY-II)

2. Condition/action, with a two-part condition consisting of a trigger and a test. The trigger provides a quick preliminary test of KS relevance. The test specifies all other prerequisites of applicability. (OPM)

3. Precondition/action in a single module, where the precondition is a trigger for a set of rules. The precondition is a list of token pairs, each pair being an event-type's name and a modifier (new/old/modified). The action component is a set of rules. Upon "focus" on an event, KSs whose precondition contains the event type are activated. Detailed tests of applicability of knowledge occur in the rules themselves. (HASP/SIAP, TRICERO)

4. Set-of-rules without preconditions, i.e., non-self-selecting KSs. (CRYSALIS)

Certain information about knowledge sources was needed globally in the system, including conditions for operation, required data types, and output data type(s). A list of all available KSs was required by the control loop to use for initiating checking. A method was sought to represent each KS as a single global object for checking, while allowing each to have several copies of itself working in parallel on different data.

Our goal was the most flexible combination of checker and actor components satisfying the criteria listed above. Based on the literature, the checker and actor components were separated and treated as two distinct procedural sets. Although the notion of a trigger is of interest, it was perceived to be more an issue of control (e.g. establishing a "focus") than a necessary component of our knowledge sources. It may be appropriate to implement this feature in the future.

We decided to construct a single knowledge source flavor built from separate **CHECKER** and **ACTOR** components (see Figure 4).[6] The **CHECKER** component would contain the condition to be tested during checking and the checking methods. The **ACTOR** component would contain methods for the KS to invoke itself and to act. The top-level knowledge source flavor (**KS-1**) would contain data only, such as a handle for the function specifying the action to be performed by the KS and descriptions of input

---

5. Reviewed in [9].

6. Terms such as object, method, instance, instantiation and flavor are technical terms from object-oriented programming. An object can be viewed as a software construct which bundles data and procedures ("methods") together. Objects interact via methods or message-passing and their data are local. A flavor is a template for a class of objects, specifying the methods and data structures common to all members of the class. Instantiation is the process of creating an object from a flavor description. Instances are the objects themselves and instance variables are data local to each instance. Flavors can be defined to include other flavors in an inheritance hierarchy. Instances of the resulting flavor "inherit" all methods and instance variables of included flavors.

and output data locations. After checking, eligible KSs would assemble descriptions of themselves and add them to a list on the blackboard. This structure allows the control loop to simply ask every KS to check itself (via **CHECKER** methods), to pick a KS to invoke from a list, and to ask that KS to invoke itself (via **ACTOR** methods).[7] Each active KS would be a *process*,[8] created by **ACTOR** methods. As many KSs as desired (even those running the same function) could then be working concurrently. **CHECKER** methods would include a check of active processes in order to avoid duplicate KS invocations. **ACTOR** methods would be responsible for requesting additional data to meet requests.

This representation promised to meet all of the criteria stated earlier. The KSs were clearly defined (each was an object), somewhat self policing (making control simpler), and independent. In addition, they could operate concurrently via processes.

3.4 Control

The issue of control was recognized to be a very important one in the development of a BBA. In the absence of clear guidelines for an efficient yet flexible control, we let the problem suggest a sufficient control structure for illustrating the flexibility of the blackboard architecture. Control, as a research issue, was left for future improvements, since it would require substantial work to develop correctly.

In general, the following features were considered desirable in a control loop:

- flexibility (ability to alter focus)
- ability to enhance processing priorities
- ability to add/remove KSs from consideration
- ability to invoke one/several KSs per cycle
- ability to change default data as required
- ability to invoke periodic blackboard purging

Of these features, none involving "focus of attention" or utilization/alteration of priorities have been implemented. Instead, a simple check-choose-invoke loop was built. The actual choice of knowledge sources to invoke (from among those eligible) was arbitrary, based simply on the ordering of the list.

---

7. Separate flavor definitions for each knowledge source were considered. Each **ACTOR** object would be written to a table on the blackboard. The approach was attractive in principle but very expensive. In each control cycle, many more KS objects would be created than used, incurring unnecessary computational overhead.

8. Processes on the Symbolics are tasks which run quasi-concurrently by taking a share of each time segment until the task is completed or halted. The "task" can consist of running a particular function.

4

IMPLEMENTATION OF BLACKBOARD ARCHITECTURE

4.1 The Blackboard

The blackboard is implemented as a hierarchy of nested hash-tables, shown in Figure 2. The global variable *bb* is a hash table containing the entire database. *bb* contains two keyed elements, each of which is a section of the blackboard. The elements are keyed "hi" and "lo", and are individual hash-tables. Each of these tables (referred to as *levels*) itself contains a set of keyed hash-tables as elements. The keys of these hash-tables refer to a *type* of data, thus the elements of the *level* tables are referred to as *type* tables. The *type* tables contain actual data about the system and the state of the problem-solving process.

The table structure organizes data in the blackboard. The hi and lo tables separate control-related data from domain-related data. "Hi" contains *type* tables such as **control-specs**, **requests**, and **invokable-ks-table**, with information useful for determining how to solve the problem. "Lo", however, contains tables such as **left-edge**, **right-edge**, and **left-side**, i.e. data directly related to the problem domain. For example, **left-edge** contains a variable number of elements, each of which is a description of a left-edge derived from a particular image.

Within each *type* table, data are organized by key. For domain data, the key almost always corresponds to the time tag of the image from which the data were derived. Other keys are usually descriptive, such as **sensor-1-cycle-time** or **'default-data-driven-priority**.

This organization of the database allows rapid location of any type of data with the efficient hashing algorithms of Symbolics Common Lisp, regardless of how large the tables grow. The hash-table software also provides internal table-locking routines, avoiding low-level read/write conflicts. With this implementation, all but one of the desired blackboard qualities were achieved (see Section 3.2). The last of the desired features, namely the ability to store and retrieve the entire database, is possible with this structure, but has not yet been investigated.

4.2 Blackboard Objects and Methods

Blackboard objects are the framework for all data kept on the blackboard and were designed for simple placement and retrieval. A hierarchy of flavors[9] was

---

9. See footnote 6.

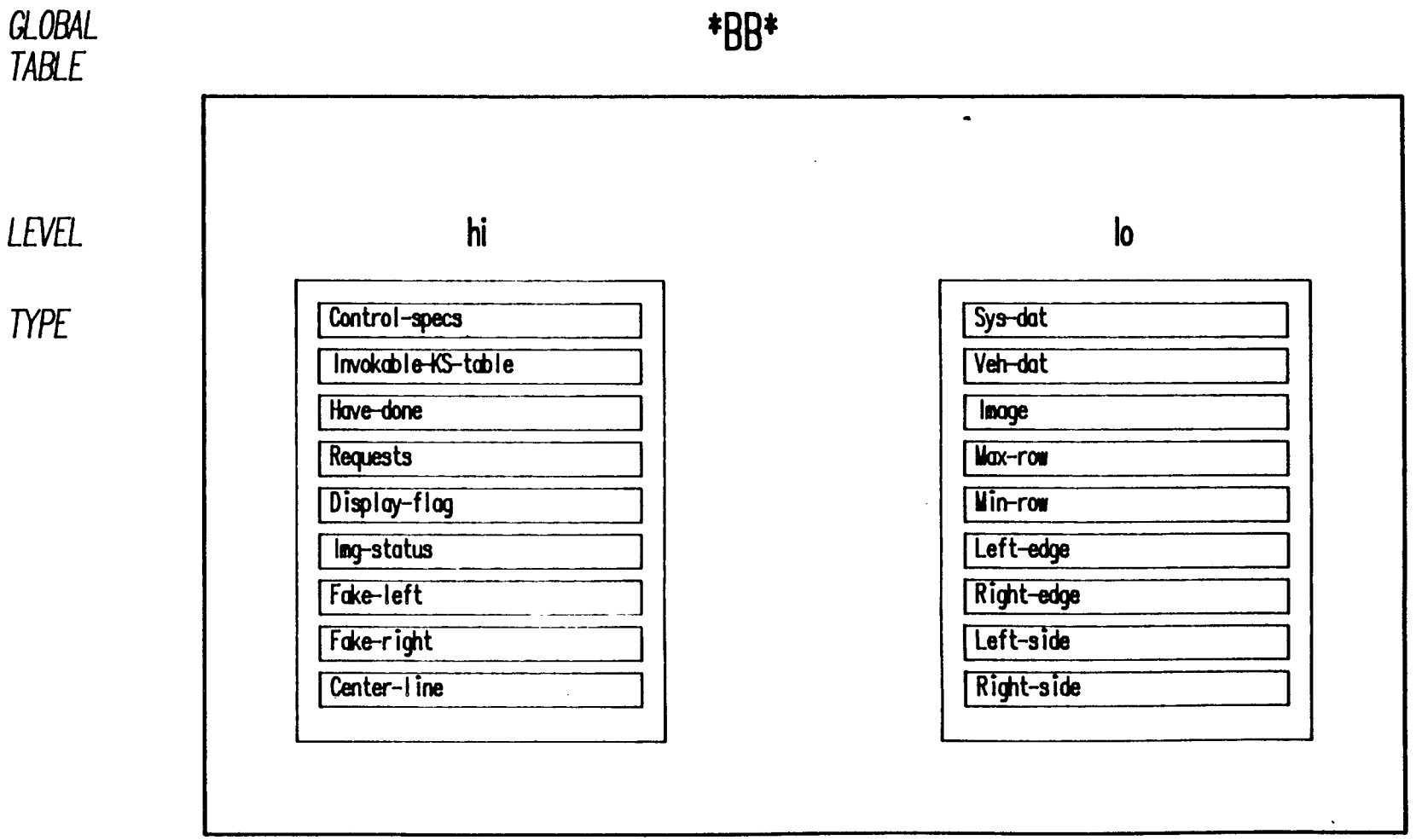


Figure 2. Organization of the Blackboard Database.

implemented (Figure 3) which puts elements common to all blackboard objects into a single base flavor, **IDENTITY**. Its instance variables contain the information necessary to add, find, and remove an item from the database: *level*, *type*, *key*, *display*, and *create-time*. *Level*, *type* and *key* define the placement of the object within the database; *display* stores a handle for the display object (if any) associated with the blackboard object and *create-time* indicates when the object was instantiated.

The elegance of the flavor decomposition is seen in the definition of the methods of the blackboard objects. Since **IDENTITY** contains all the data necessary to place an object in the blackboard, and since **IDENTITY** is included in each blackboard object, one set of methods is used to add, locate, and remove any flavor of blackboard object. These methods are aptly named **add-to-bb**, **locate**, and **rem-from-bb**. Two other methods of **IDENTITY**, **place-icon** and **present-icon**, are concerned with display and are discussed further in Section 4.5.

The **locate** method uses the *level* and *type* information in **IDENTITY** to return a handle for the table in which the blackboard object resides. **Add-to-bb** uses *level*, *type*, and *key* data to place the object in the correct table with the correct key. It also displays the object's icon and saves a handle for the icon in the *display* variable. **Rem-from-bb** removes the object from the blackboard and also from the display if applicable (*display* is not-NIL).

The **IDENTITY** flavor is not itself instantiable, but forms the basis for all instantiable blackboard objects. Three flavors of blackboard objects were defined: **DATA-OBJECT**, **REQUEST** and **KS-DESCRIPTION**. **DATA-OBJECT** is the primary flavor. It consists of the **IDENTITY** flavor and one instance variable: *item*. *Item* is the data variable for the object, and may contain any Lisp object as its value. **DATA-OBJECT** is used to record all initial data and current hypotheses.

REQUEST and **KS-DESCRIPTION** define blackboard objects which store information about requests for data and about invokable knowledge sources. The data they contain could have been placed in the *item* variable of a **DATA-OBJECT**, but were more clearly represented by the following implementation. **REQUEST** is comprised of the **IDENTITY** flavor and the instance variables: *priority*, *quantity*, *req-type*, *requestor*, and *input-key*. *Req-type* and *input-key* identify the type and time of the data requested; *requestor* stores the name of the requesting KS. *Priority* reflects how fast the result is required. *Quantity* was intended to identify the number of consecutive (in time) results desired, but has never been used. **KS-DESCRIPTION** consists of the **IDENTITY** flavor and the instance variables *KS-ref*, *input-key*, *requestor*, *priority*, and *data*. *KS-ref* is a handle for the knowledge source being described in the object; *input-key* specifies the time-frame of data for which the KS is invokable. *Requestor* is the KS object which requested the data produced by *KS-ref* (if any). *Priority* is intended to reflect the urgency with which the KS should be invoked, but is currently unused. The last variable, *data*, indicates if all of the required input data exist on the blackboard for *KS-ref* to operate.

4.3 Knowledge Sources (KSs)

Knowledge sources were defined corresponding to each of the sub-problems identified in Section 3.1. Like blackboard objects, knowledge sources were implemented using a flavor hierarchy, as shown in Figure 4. **CHECKER** and **ACTOR** are non-instantiable base flavors and are included in **KS-1**, the instantiable knowledge source flavor. The main purpose for **CHECKER** and **ACTOR** is to define the necessary

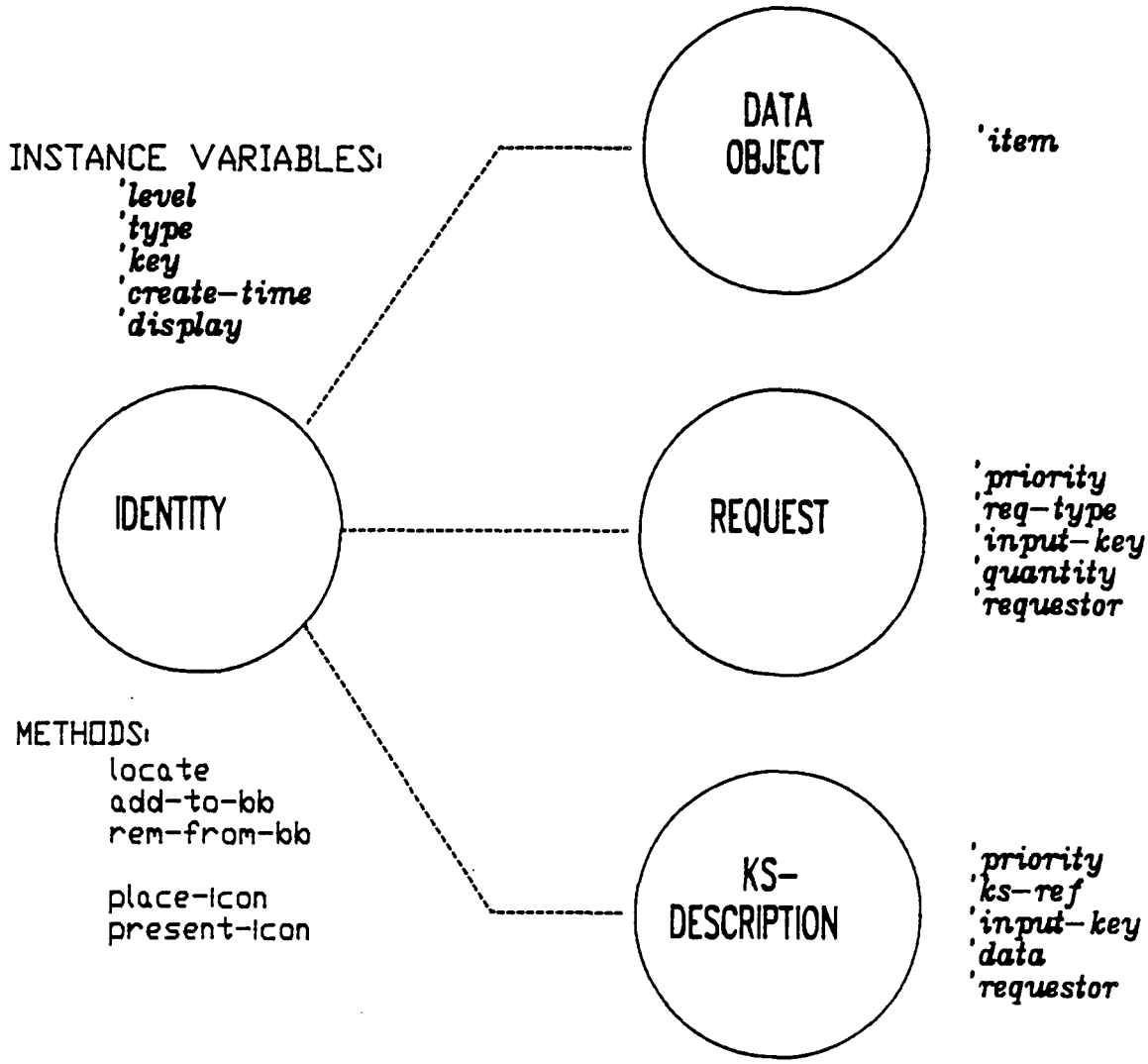


Figure 3. Flavor Hierarchy of Blackboard Objects.

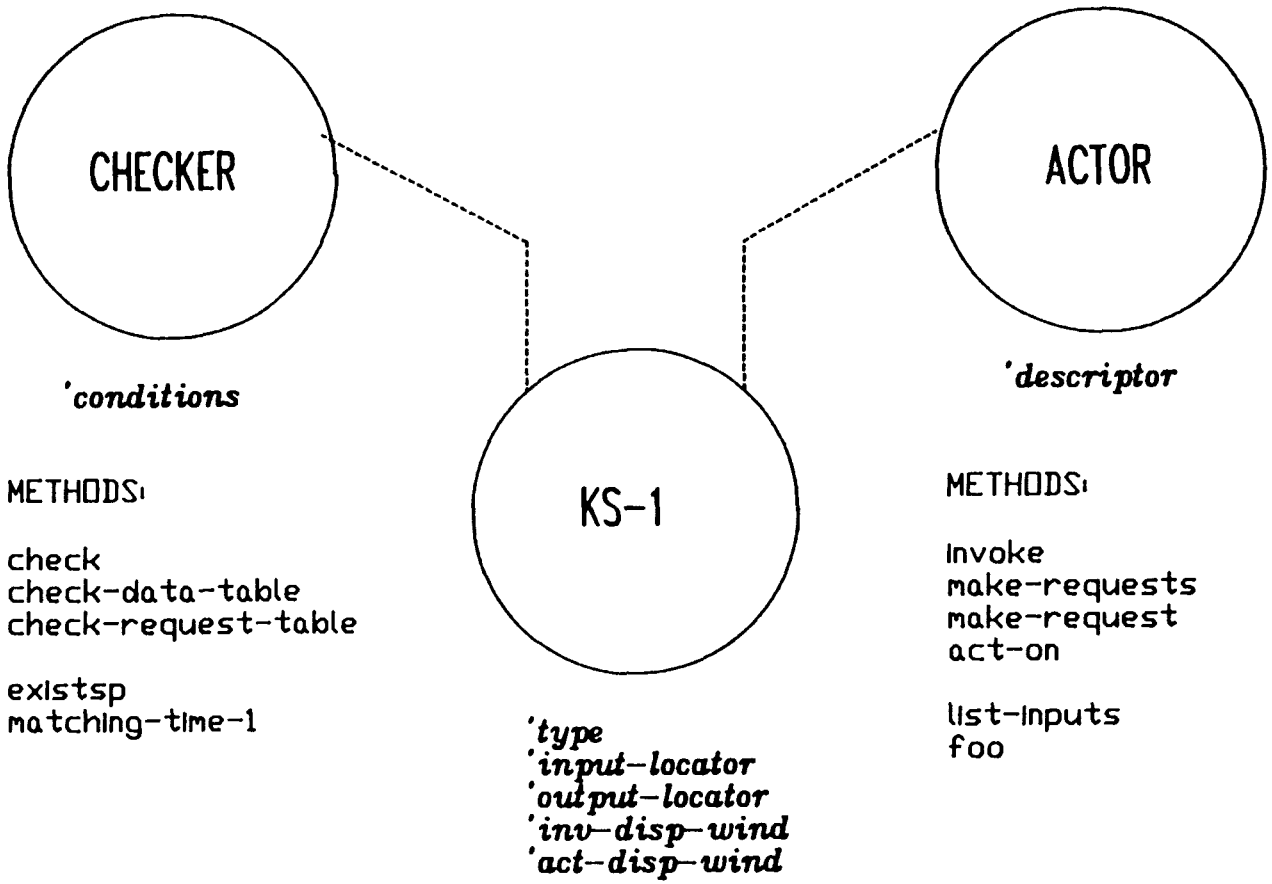


Figure 4. Flavor Hierarchy of Knowledge Sources.

methods, as discussed in Section 3.3. Subdividing the main flavor in this way permits future addition of other KS flavors, by including altered checker or actor flavors and methods.

CHECKER has one instance variable, *conditions*, which plays a very important role in the checker methods. **ACTOR** also has one instance variable, *descriptor*, which is currently unused. **KS-1** is built from CHECKER and ACTOR and has the additional instance variables: *type*, *input-locator*, *output-locator*, *inv-disp-wind*, and *act-disp-wind*. *Type* holds the name of function which performs the action of the KS. *Input-locator* and *output-locator* store level and type information about the required input data and the resulting output data, respectively. *Inv-disp-wind* and *act-disp-wind* are used in the display.

Exactly one **KS-1** is instantiated for each KS within the BBA. This object is always in existence and has its own methods for checking and invoking, through the flavor hierarchy. In effect, the **KS-1** object is used as a template by the **ACTOR** methods. This limits the space taken by KS definitions, while allowing many copies of each KS to be made (in the form of **KS-DESCRIPTION** objects) when needed.

If a set of required inputs with matching keys exists for a particular **KS-1**, and the corresponding outputs do not, then the KS can be invoked in a data-driven fashion. If a request is on the blackboard for the output of a certain knowledge source, then it can be invoked in a request-driven fashion regardless of the existence of its required inputs. If a knowledge source has been requested and some or all of its required input data are missing, requests for missing data are placed on the blackboard. These requests in turn trigger invokation of other knowledge sources. Requests propagate through the system until inputs for the latest request are found. Results then propagate in the reverse direction until all requests are satisfied.

The methods defined in **CHECKER** are **check**, **check-data-table**, **check-request-table**, **match-request-time-1**, and **existsp**. The first three are the main checking algorithms. **Check** applies **check-data-table** to the elements of the table specified by the first element of the *input-locator*, if any, and then applies **check-request-table** to each element of the blackboard's request table. **Check-data-table** performs the checking necessary to determine if a KS is data-driven invokable. For each element of the input table, this method checks if the KS has already been invoked for that key, if the result from this KS for the key already exists, and if *conditions* are either non-existent or are met. If the result doesn't already exist, an equivalent KS is not already processing, and *conditions* are met, then the method checks for the existence of all secondary input data (using **existsp**). When all other required data exist, a **KS-DESCRIPTION** is created and placed in the invokable-ks-table of the blackboard. A **KS-DESCRIPTION** is therefore created for every knowledge source-data pair; these are data-driven invokable KSs.

Check-request-table checks the *type* variable of each *request* object against the data type in the KS's *output-locator*. If the types match, this method attempts to find a more exact time-frame reference (via **match-request-time-1**). If it cannot be matched, or if some required data are missing from invocation of this KS, the *data* variable of the created **KS-DESCRIPTION** object is set NIL. A **KS-DESCRIPTION** is always written when the types match, to insure that all requests can be serviced. These are request-driven invokable KSs.

Match-request-time-1 uses initial data about the sensor's cycle time to find the closest match between a time reference for which a result was requested, and time references for which data exist. **Existsp** uses the *input-locator* data of the KS to determine whether or not all required input data exist on the blackboard.

The methods defined in **ACTOR** are **invoke**, **make-requests**, **make-request**, **act-on**, **list-inputs**, and **foo**, of which the primary method is **invoke**. After selecting from the eligible KSs (**KS-DESCRIPTIONs**), the control structure calls the **Invoke** method of the corresponding **KS-1**. This method first places a marker on the blackboard to indicate that the particular KS has been invoked and is operating on data with the corresponding *key*. The marker is removed by the KS function after it is finished. **Invoke** then looks at *requestor* to determine if the KS was requested. If not, the KS is invoked immediately via **act-on**. **Act-on** creates a process to run the KS function (specified by the **KS-1** instance variable *type*) on the data listed by **list-inputs**. **Act-on** assumes all data in the input list are currently on the blackboard. **List-inputs** makes a list of all data associated with an *input-locator*, and a *key*.

If a KS was requested, the corresponding request is removed from the blackboard. Next, the existence of all input data is checked. If all data exist, the KS is invoked via **act-on**. If not, requests for the missing data are placed on the blackboard. **Make-requests** uses the method **make-request** for each missing datum. **Invoke** then creates a process to run the method **foo**, which is just a busy wait for the missing data.

4.4 Control

Only a basic level of control has been implemented. Major concerns were to show:

- A preference to invoke emergency KSs as soon as possible.
- A preference to invoke KSs which signal other important conditions .
- An ability to prioritize KSs.
- An ability to concentrate processing efforts on a specific task.

The first two were implemented by breaking the check-invoke cycle into three steps, corresponding to three lists in the blackboard *control-specs* table. These lists are keyed *emergency-cases-list*, *notification-KS-list*, and *available-KS-list*. Each list contains the *KS-descriptions* of the appropriate sublist of existing KSs. In the check-invoke loop, the emergency cases are checked first, and ALL invokable KSs found are immediately invoked. Next, the notification KSs are checked (these are the loss of left or right or both side of the road). Again, EVERY invokable KS found is promptly invoked. Finally, the other KSs, both data and request driven, are checked and exactly one is invoked, that being the first found in the list. This scheme ensures that flags of impending disaster or emergency handling KSs are promptly invoked, and that repetitive flags and monitors, like the audible *both-missing* KS, are not invoked to the exclusion of other invokable data and request driven KSs.

Another desired feature of the control is that any emergency KS invoked is done so with a priority higher than that of the normal KS. If the news is "bad" i.e. if the KS discovers a situation of impending disaster, the system should be all but halted until a go-ahead is indicated via the keyboard. This would indicate the ability of the architecture to alter its behavior to fit an extreme situation. In a full vehicle planning system, the priority might be placed on exploratory scanning and processing as opposed to keyboard input, but the point is to be able to focus processing activities in an extreme situation.

4.5 Display

The display has two distinct functions, to show the processing being accomplished and to show the operation of the system. The display is composed of three distinct areas: processing bar graphs, process-icons, and warning messages. A schematic of the display is given in Figure 5.

Bar graphs represent the invokable-invoked portion of the control in the following manner. When a KS is added to the *invokable-KS-table*, the corresponding bar in the graph (if one exists) is incremented. Upon invocation of a specific KS, the corresponding bar in the "invoked" or "active" bar graph is incremented. The active graph is decremented appropriately upon completion of each KS process. The invokable KS bar graph is cleared after each check-invoke cycle to highlight the action of the control loop.

The process-icon section presents the operation of the BBA in a scrolling 2d representation. Time increases from top to bottom of this section. Each raw sensor image is displayed on the left, in the vertical position corresponding to its time of acquisition.[10] Processing steps (KSs) are represented by icons at specific positions across the display with the level of abstraction increasing from left to right. A running figure indicates a process that is currently acting. Upon completion of the process, the runner is replaced by an icon representing the result obtained by the KS. The runner is drawn in red if the KS has been invoked as a result of a request. This means that if a KS is invoked due to request and must in turn request other data, red running figures progress to the left (toward more basic information). As requests are fulfilled, white icons progress to the right, replacing the red runners.

The third (bottom) portion of the display is reserved for flags and messages. This portion of the screen is used for printing warning messages which flag the failure to detect one or both road sides, and to prompt for a continuation after an indication of potential disaster (e.g. loss of both road sides or detection of a possible obstacle in front of the vehicle).

Two of the display methods are methods of the **IDENTITY** flavor. Briefly, these methods are **place-icon** and **present-icon**. **Place-icon** uses the type information existing in all objects of the blackboard to determine the screen position of the icon to represent the object. In the display, column is determined by the type, and row is determined by the key or time stamp. **Place-icon** determines the proper row and column, and passes through information on whether or not the object was requested. The last thing **place-icon** does is to call **present-icon**. **Present-icon** uses the objects type information to determine the proper icon to draw. It draws the icon to the row and column passed to it, and draws it in red if the object was requested. The icon is drawn as a *presentation*, a LISP concept which allows easy definition of mouse sensitivity for displayed objects.

4.6 Sensor Simulation

Sensor images were stored in individual files on disk. As discussed above (Section 3), images were of two types: raw range (angle, angle, range) and pre-processed

---

10. See Section 4.6.

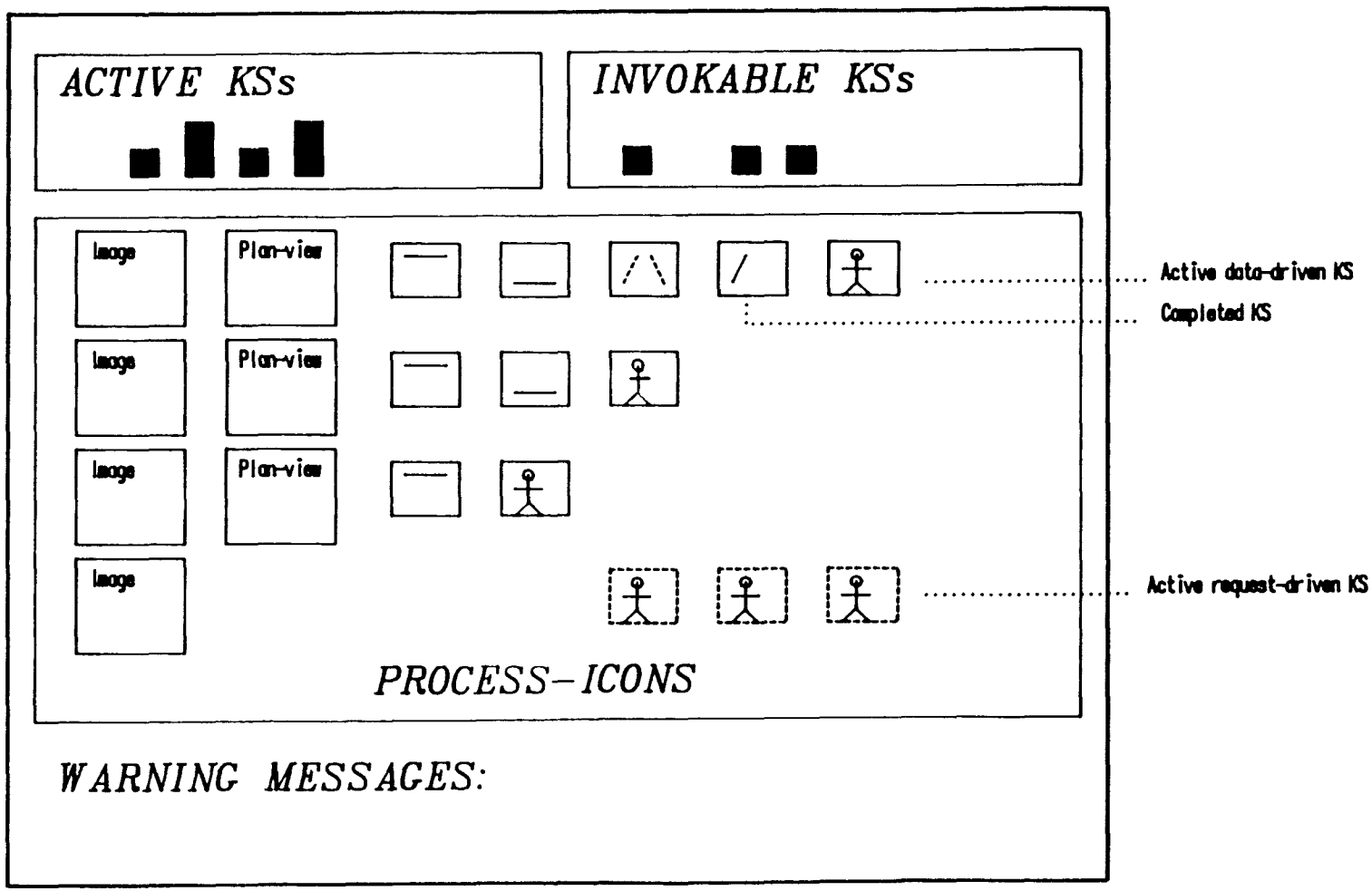


Figure 5. Schematic of Display

plan views (Cartesian x-y-z). To mimic the operation of a continuous sensor, a process was created which read one image of each type in sequence and assigned a time tag. Images and their time tags were stored arrays in one of two lists termed *image-queue1* (raw) and *image-queue2* (pre-processed). A knowledge source, the *Dispatcher*, was defined which moved images from the image-queues to the appropriate location in the blackboard database. This knowledge source was invokable whenever a new image appeared on either queue.

4.7 Organization of Code

Listings of the code are included as an Appendix. The first listing is of a file containing all statements required to compile and load the system, COMPLOAD.TAC. The definition of the blackboard database and blackboard objects is in the file BBDEF.TAC. Knowledge source flavors are defined in KSDEF.TAC, which also includes code to instantiate a single copy of each knowledge source object in the system. The actual functions for each KS are located in KSFUNS.TAC. **CHECKER** and **ACTOR** methods are defined in CHECKER.TAC and ACTOR.TAC, respectively. CONTROL.TAC defines the control loop.

Two files contain the majority of the code for the display, BBDEF.TAC and BBDISP.TAC. Some of the display is in the BBDEF file because two of the display methods are methods of the **IDENTITY** flavor. The other file, BBDISP, contains the setup of the windows on the color screen, the increment and decrement methods for the bar graph windows, and the methods to write textual messages to the screen. The sensor simulation and associated code is defined in SENSOR2.TAC, while miscellaneous functions to interact with the BBA are defined in TESTBASE.TAC.

5

SUMMARY

The following paragraphs are a summary of the blackboard architecture development in practical, operational terms. Following this summary are a few brief paragraphs which compare and contrast this system with documented BBAs. Finally, recommendations for future research are outlined.

The blackboard architecture (BBA) accepted a set of images as initial input data. The image data consisted of actual sensor images collected while driving a vehicle. Characteristics of the images included: a section of straight road, a curve, and a wide open area such as a parking lot. The images were made available to the system as fast as they could be read off disk, in order to simulate continuous sensing. Once into the BBA, each image set in motion the default data-driven processing sequence, described in more detail below. This sequence resulted in either identification of the sides of the road or a highly visible indication that they could not be found.

The default data-driven sequence included the following processing steps:

- Convert the sensor image to plan-view (x y z)
- Delimit the usable portion of the image
- Mark points in the image which appear to be edges
- Extrapolate a line, if possible, for the left side of the road
- Extrapolate a line, if possible, for the right side of the road
- Notify the user if the left road side cannot be found
- Notify the user if the right road side cannot be found

Unless other input to the system occurred, such as a user-supplied request for a particular result, these steps proceeded in a sequential, data-driven fashion for each image as it entered the system.

The potential flexibility of the BBA was demonstrated by two distinct means. First, although each image was processed sequentially in the default, data-driven mode, nothing prevented processing of more than one image at a time. Demonstration of concurrent processing of several images, in both similar and different stages of processing, was given.

Second, the use of requests was shown. Requests could be entered at the keyboard for any of the data producible by the above processing steps. If the requested datum could not be determined due to lack of input data, the request would propagate through the system, i.e. the process invoked by the original request would post requests within the BBA for the input data it required. At the same time, the system was still processing other sensor images in its data-driven default mode.

Thus, the final BBA demonstration showed the capabilities of the system to handle real data, to process data concurrently (whether data-driven, request-driven, or

both), to propagate requests internally when required, and to monitor for exceptional situations (loss of one or both road sides).

Data organization in the BBA developed for TACOM is consistent with those reported in the literature. A hierarchical data structure was not implemented, but data were sub-divided and grouped based on level of abstraction and control/domain considerations. Our implementation of knowledge sources may be unique; insufficient detail is available in literature descriptions to be sure. They do consist of checker and actor components, as do most of those documented. No trigger was been implemented, but pre-conditions exist above and beyond input data requirements. The control structure implemented is similar to the most basic of those found in the literature. More extensively developed control structures have been documented, but no globally acceptable one is known to exist. It is feared that the control is very much dependent on the specific problem, causing each BBA's control structure to vary considerably.

There are many potential areas of future work and possible improvements on the TACOM BBA. As previously mentioned, the control structure is an area of research itself. No attention has been given to issues of concurrency--interference between knowledge sources. This must be addressed as the KS structure makes parallel processing an inherent part of the BBA. The request mechanism of our BBA requires improvement; no allowance has been made for "hanging requests"; if a request cannot be answered, it remains in a busy wait, using precious system time. The failure to provide requested data must be recognized and propagated through the system to the originating requestor.

Another primary issue, that of time, has not been adequately handled in our system. Time plays a very important role in the validity of sensory information and results derived from these data. Time also is an issue in emergency situations, where a less costly (in terms of time to solution) approach to a decision may be necessary to avoid disaster.

These are some of the principal outstanding problems with the current BBA. It is hoped that future efforts could be directed to improvements in these areas.

6

BIBLIOGRAPHY

[1] Blidberg, D. Richard, "Time-Ordered Architecture for Knowledge-Based Guidance of an Unmanned Untethered Submersible", IEEE Oceans, 1984, pp.571-575.

[2] Chappell, Steven G., "A Prototype Trajectory Planning system for an Autonomous Vehicle", Blackboard Workshop at Carnegie-Mellon University, June 12-13, 1986 (preprint).

[3] Erman, Lee D., et al., "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty", Computing Surveys, Vol. 12, No.2, June 1980, pp. 213-252.

[4] Franklin Jr., Robert F. and Hoogterp, Francis B., "Sensing and Planning For Autonomous Road Following". Eighth Annual Symposium on Ground Vehicle Signatures, Keweenaw Research Center, Houghton, MI. Vol. 1, 226-237 (1986).

[5] Gilmore, John F., and Semeco, Antonio C., "Terrain Navigation Through Knowledge-Based Route Planning", International Joint Conference on AI, August, 1985.

[6] Gilmore, John F., and Semeco, Antonio C., "Autonomous Route Planning Through Non-uniform Terrain", IEEE Computer Vision and Pattern Recognition, 1985.

[7] Harmon, S. Y., Bianchini, B. E. Pinz, "Sensor Data Fusion Through a Distributed Blackboard", IEEE 1986 Conference on Robotics and Automation, pp. 1449-1454.

[8] Hayes-Roth, Barbara, "A Blackboard Architecture for Control", Artificial Intelligence 26 (1985), pp. 251-321.

[9] Nii, H. Penny, "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures:, AI, Summer, 1986, pp. 38-53.

[10] Nitao, John J., and Parodi, Alexandre M., "An Intelligent Pilot for an Autonomous Vehicle System", Proceedings 1985 IEEE Second Conference on Artificial Intelligence Applications, pp. 176-183.

[11] Payton, David W., "An Architecture for Reflexive Autonomous Vehicle Control", Proceedings 1986 IEEE International Conference on Robotics and Automation, pp. 1838-1845.

[12] Pearson, Glen, and Kuan, Darwin, "Mission Planning System for an Autonomous Vehicle", Proceedings 1985 IEEE Second Conference on Artificial Intelligence Applications, pp. 162-167.

[13] Shafer, Steven A., Stentz, Anthony, and Thorpe, Charles E., "An Architecture for Sensor Fusion in a Mobile Robot", Proceedings 1986 IEEE International Conference on Robotics and Automation, pp. 2002-2011.

[14] Skillman Jr., Thomas L., "Blackboard Based Control of an Autonomous Space Craft", Blackboard Workshop at Carnegie-Mellon University, June 12-13, 1986 (in print).

[15] Symbolics System Documentation, Volume 9, "Networks", June, 1986.

[16] Yang, Ju-Yuan David, Huhns, Michael N., and Stephens, Larry M., "An Architecture for Control and Communications in Distributed Artificial Intelligence systems", IEE Transactions on Systems, Man, and Cybernetics, May/June 1985, pp. 316-326.

7

APPENDIX

```
; -*- Mode:  lisp; Base: 10; Package: CL-USER; Syntax: Common-Lisp -*-

(compiler:compile-file-load "vino:>sprout>current-bba>bbdef")
(compiler:compile-file-load "vino:>sprout>current-bba>bbdisp")
(compiler:compile-file-load "vino:>sprout>current-bba>sensor2")

(compiler:compile-file-load "vino:>sprout>current-bba>ksfuns")

(compiler:compile-file-load "vino:>sprout>current-bba>ksdef")
(compiler:compile-file-load "vino:>sprout>current-bba>checker")
(compiler:compile-file-load "vino:>sprout>current-bba>actor")

(compiler:compile-file-load "vino:>sprout>current-bba>testbase")

(compiler:compile-file-load "vino:>sprout>current-bba>control")
```

```
; -*- Mode:  lisp; Base: 10; Package: CL-USER; Syntax: Common-Lisp -*-

;******* FILE:     BBDEF.LISP ***************************************
;******* Last-edit:  06/01/87  ***********************************

;;;This file, bbdef.lisp, contains the definition of the blackboard
; data structure, the flavor definitions for the blackboard data
; objects, and some initializations to the blackboard.


;;; *bb* is the blackboard database structure, as given below.  The
; cases which have :test set to equalp are those tables whose keys
; are lists.
;;;

(DEFVAR *bb*                                    ; this is the global
        (MAKE-HASH-TABLE                        ; database, consisting
          :SIZE 2                               ; of 2 sub-tables
          :INITIAL-CONTENTS
          '(hi                                  ; 'lo and 'hi refer
             ,(MAKE-HASH-TABLE                  ; to a vague data
                :INITIAL-CONTENTS               ; hierarchy
                '(requests ,(MAKE-HASH-TABLE
                              :TEST 'EQUALP )
                  invocable-ks-table ,(MAKE-HASH-TABLE
                                        :TEST 'EQUALP )
                  have-done ,(MAKE-HASH-TABLE
                                        :TEST 'EQUALP )
                  control-specs ,(MAKE-HASH-TABLE)
                  display-flag ,(MAKE-HASH-TABLE) ;used by mouse handler
                  fake-left ,(MAKE-HASH-TABLE)
                  fake-right ,(MAKE-HASH-TABLE)
                  center-line ,(MAKE-HASH-TABLE)
                  img-status ,(MAKE-HASH-TABLE))) ;final marked images

            lo
             ,(MAKE-HASH-TABLE
                :INITIAL-CONTENTS
                '(image     ,(MAKE-HASH-TABLE)
                                                ;raw images from dispatcher
                  max-row ,(MAKE-HASH-TABLE)
                  min-row ,(MAKE-HASH-TABLE)
                  veh-dat ,(MAKE-HASH-TABLE)
                                                ; basic vehicle info
                  sys-dat ,(MAKE-HASH-TABLE)
                                                ; basic system info
                  left-edge  ,(MAKE-HASH-TABLE)
                  right-edge ,(MAKE-HASH-TABLE)
                  left-side  ,(MAKE-HASH-TABLE)
                  right-side ,(MAKE-HASH-TABLE))))))


;;;What follows are the definitions of the objects which will exist
; on the blackboard.  Methods to find, add, and remove objects are
; defined on these flavors.
;;;


(DEFFLAVOR identity                             ;comp. flavor for all bb objects
        (level                                  ;level of bb tables
         type                                   ;local table in level
         key                                    ;key in local table
```

```
        display                                 ;presentation handle
        (create-time (ZL:TIME)))                ;object create time
        ( )                                     ;no component flavors
  (:CONC-NAME NIL)
  :ABSTRACT-FLAVOR
  :INITABLE-INSTANCE-VARIABLES
  :WRITABLE-INSTANCE-VARIABLES )


(DEFFLAVOR data-object                          ;generic data object for bb
        (item)                                  ;data variable
        (identity )                             ;component flavor
  :INITABLE-INSTANCE-VARIABLES
  :WRITABLE-INSTANCE-VARIABLES
  (:CONC-NAME get- ))


(DEFFLAVOR request                              ;request for data from bb object
           (priority                            ;priority of the request object
            quantity                            ;used for multiple image request
            req-type                            ;what type of data requested
            requestor                           ;who wrote the request object
            input-key )                         ;used to identify time frame
            (identity)
    :INITABLE-INSTANCE-VARIABLES
    :WRITABLE-INSTANCE-VARIABLES )


(DEFFLAVOR KS-description
        ( ks-ref                                ;whose description
          input-key                             ;input location, if exists
          (requestor       NIL)                 ;ks-ref or nil if data driven
          priority                              ;any available priority info
          data )                                ;flags existance of input data
        ( identity )                            ;component flav w/ methods
  :INITABLE-INSTANCE-VARIABLES
  :READABLE-INSTANCE-VARIABLES
  (:CONC-NAME NIL ))

;;; a couple of display methods.....
; These are defined here because they are methods on identity.
;;;


;;; PLACE-ICON
; This method determines the desired position of the icon (representing
; SELF) in the icon-display window.  The type of the object determines
; its horizontal position, and the key (time reference) determines its
; vertical position.  Req-flag is passed through this method as an
; indication of whether or not the object has been requested.  If it
; has, the color of some icons is changed.  This method calls the method
; PRESENT-ICON which actually draws the icon to the desired position on
; the display window.
;;;

(DEFMETHOD ( place-icon identity ) (req-flag)
  (LET ((type (type self))                      ;the object type
        (row)                                   ;abs row position PIX
        (col))                                  ;abs column position PIX
    (IF (EQUALP type 'have-done)                ;this is a working one
        (SETQ type (FIRST (key self))))         ;get actual object type
    (COND ((OR (EQUALP type 'image)             ;image being written to bb
               (EQUALP type "dispatcher))       ;image being sought
```

```
          (SETQ col 150))
         ((OR (EQUALP type "find-max-row)      ;working KS
              (EQUALP type 'max-row))          ;result being written
          (SETQ col 300))
         ((OR (EQUALP type "find-min-row)
              (EQUALP type 'min-row))
          (SETQ col 350))
         ((OR (EQUALP type "find-edges)
              (EQUALP type 'left-edge)
              (EQUALP type 'right-edge))
          (SETQ col 400))
         ((OR (EQUALP type "find-left-side)
              (EQUALP type 'left-side))
          (SETQ col 450))
         ((OR (EQUALP type "find-right-side)
              (EQUALP type 'right-side))
          (SETQ col 500))
         (T NIL))                              ;if none of above, do naught
    (IF (NULL col) NIL
        (IF (LISTP key )                       ;may have to parse key
            (SETQ row (+ (* 3 (SECOND key)) 50));vertical spacing of 150 PIX
            (SETQ row (+ (* 3 key ) 50)))
    (present-icon SELF col row req-flag))))    ;draw the icon


;;;PRESENT-ICON
; This method contains the code to actually draw the proper icon to the
; desired screen location.  The icon to be drawn depends on the object's
; type.  All objects of type 'have-done are representations of KSs in
; action.  These get represented by a running figure on the color screen.
; The runner is drawn in red if the KS is request-driven.
;;;


(DEFMETHOD ( present-icon identity ) (left top req-flag
                                      &OPTIONAL (stream '*icon-disp*))
  (DECLARE (SPECIAL *icon-disp* *img-disp*))
  (LET ((presentation-list)
        (tmp-alu *demo-window-alu*)
        (basic-presentation                    ;this is the square box which
          '(DW:WITH-OUTPUT-AS-PRESENTATION
                                               ;surrounds the icons
             (:SINGLE-BOX T
              :STREAM ,stream
              :TYPE 'instance
              :OBJECT ,self)
             (SEND ,stream :DRAW-RECTANGLE 40 40 ,left ,top COLOR:ALU-0S)
             (SEND ,stream :DRAW-LINE ,left ,top ,(+ left 40) ,top)
             (SEND ,stream :DRAW-LINE ,(+ left 40) ,top ,(+ left 40) ,(+ top 40))
             (SEND ,stream :DRAW-LINE ,(+ left 40) ,(+ top 40) ,left ,(+ top 40))
             (SEND ,stream :DRAW-LINE ,left ,(+ top 40) ,left ,top))))

    (IF req-flag                               ;the object was requested
        (SEND tmp-alu :SET-FILL-DATA           ;so draw icon in red
              *danger-color*)
        (SETQ tmp-alu (SEND *icon-disp* :CHAR-ALUF)))

    (COND ((EQUALP (type SELF) 'image)         ;draw a big box so it's mouse
           (LET ((it (image (get-item SELF))))  ;sensitive over the image
             (SETQ presentation-list
                   '(DW:WITH-OUTPUT-AS-PRESENTATION
                      (:SINGLE-BOX T
                       :STREAM ,stream
```

```
                :TYPE 'instance
                :OBJECT ,self)
               (SEND ,stream :DRAW-LINES TV:ALU-XOR ,left
                     ,(- top 50) ,(+ left 120) ,(- top 50)
                     ,(+ left 120) ,(+ top 70) ,left
                     ,(+ top 70) ,left ,(- top 50))
               (SEND ,stream :BITBLT TV:ALU-SETA
                     ,(CAR (ARRAY-DIMENSIONS it))
                     ,(CADR (ARRAY-DIMENSIONS it))
                     ,it 0 0 ,left ,(- top 50))))))

((EQUALP (type SELF) 'max-row)          ;add a bar to the square box
 (SETQ presentation-list
       (APPEND basic-presentation
               '((SEND ,stream :DRAW-LINE ,(+ left 5)
                       ,(+ top 5) ,(+ left 35) ,(+ top 5))))))

((EQUALP (type SELF) 'min-row)          ;add a bar to the square box
 (SETQ presentation-list
       (APPEND basic-presentation
               '((SEND ,stream :DRAW-LINE ,(+ left 5)
                       ,(+ top 35) ,(+ left 35) ,(+ top 35))))))

((OR (EQUALP (type SELF) 'left-edge)    ;add dashed lines to the box
     (EQUALP (type SELF) 'right-edge))
 (SETQ presentation-list
       (APPEND basic-presentation
               '((SEND ,stream :DRAW-DASHED-LINE ,(+ left 10)
                       ,(+ top 5) ,(+ left 5) ,(+ top 35))
                 (SEND ,stream :DRAW-DASHED-LINE ,(+ left 30)
                       ,(+ top 5) ,(+ left 35) ,(+ top 35))))))

((EQUALP (type SELF) 'left-side)        ;add a line to the box
 (SETQ presentation-list
       (APPEND basic-presentation
               '((SEND ,stream :DRAW-LINE ,(+ left 10)
                       ,(+ top 5) ,(+ left 5) ,(+ top 35))))))

((EQUALP (type SELF) 'right-side)       ;add a line to the box
 (SETQ presentation-list
       (APPEND basic-presentation
               '((SEND ,stream :DRAW-LINE ,(+ left 30)
                       ,(+ top 5) ,(+ left 35) ,(+ top 35))))))

((EQUALP (type SELF) 'have-done)        ;add the runner to the box
 (SETQ presentation-list
```

```
                (APPEND basic-presentation
                        '((SEND ,stream  :DRAW-LINE ,(+ left 22)
                                ,(+ top 22) ,(+ left 19) ,(+ top 29) ,tmp-alu)
                          (SEND  ,stream :DRAW-LINE ,(+ left 19)
                                ,(+ top 29) ,(+ left 12) ,(+ top 27) ,tmp-alu)
                          (SEND  ,stream :DRAW-LINE ,(+ left 22)
                                ,(+ top 22) ,(+ left 24) ,(+ top 32) ,tmp-alu)
                          (SEND  ,stream :DRAW-LINE ,(+ left 22)
                                ,(+ top 22) ,(+ left 24) ,(+ top 7) ,tmp-alu)
                          (SEND  ,stream :DRAW-LINE ,(+ left 24)
                                ,(+ top 32) ,(+ left 22) ,(+ top 37) ,tmp-alu)
                          (SEND  ,stream :DRAW-FILLED-IN-CIRCLE
                                ,(+ left 24) ,(+ top 7) 5 ,tmp-alu)
                          (SEND  ,stream :DRAW-LINE ,(+ left 24)
                                ,(+ top 16) ,(+ left 12) ,(+ top 12) ,tmp-alu)
                          (SEND  ,stream :DRAW-LINE ,(+ left 24)
                                ,(+ top 16) ,(+ left 32) ,(+ top 22) ,tmp-alu)
                          ))))
        (T NIL))
    (EVAL presentation-list)))                    ;draw the figure-


;;;LOCATE
; This method uses level and type information contained in SELF to
; locate and return a handle on the hash-table within which this
; object belongs
;;;

(DEFMETHOD (locate identity) ( )
  (GETHASH type (GETHASH level *bb*)))          ;return the table object


;;;ADD-TO-BB
; This method adds SELF to the table indicated by LOCATE, and then
; calls the display method.  In this way, all displays of things on
; the blackboard are drawn as they are written to the blackboard.  An
; instance variable is set to the handle of the display so that the
; presentation can be deleted if desired.
;;;

(DEFMETHOD ( add-to-bb identity )
           (&OPTIONAL (req-flag NIL ))
  (SETF (GETHASH key (locate SELF)) SELF)       ;add object to bb
  (SETF display (place-icon SELF req-flag)))    ;***display object


;;;REM-FROM-BB
; This method removes SELF from the table indicated by LOCATE.  If the
; object being removed is one from the 'have-done table, then the
; presentation for that object is removed from the display window.  The
; restriction to that type of object may be removed, as it will be
; desirable not to have a presentation for any object that no longer exists
; in the blackboard.
;;;

(DEFMETHOD (rem-from-bb identity) ( )
  (DECLARE (SPECIAL *icon-disp*))
  (IF (AND (EQUALP type 'have-done)             ;remove needless display
           (display self))
      (SEND *icon-disp*                         ;to avoid redisplay on
```

```
            :DELETE-DISPLAYED-PRESENTATION
                                                ;scrolled back window
            display))
  (REMHASH key (locate SELF)))                  ;remove object from bb


;;;GET-TABLE
; This function is used to return a handle for a table when the level
; and type are known but no object is specifying them.
;;;

(DEFUN get-table (locator)                      ;function to access a
  (LET ((level (FIRST locator))                 ;table when no object
        (type (SECOND locator)))                ;is currently known.
    (GETHASH type (GETHASH level *bb*))))


;;;The following are some initializations of the blackboard database
; structure.  This section is intended to create those data-objects
; which are vehicle related or vehicle constant, and which will be
; required by the knowledge sources.
;;;

(add-to-bb (MAKE-INSTANCE 'data-object          ;half-width of the
                          :level 'lo            ;vehicle
                          :type 'veh-dat
                          :key 'half-width
                          :item 7 ))


(add-to-bb (MAKE-INSTANCE 'data-object          ;cycle time of the
                          :level 'lo            ;xyz sensor.
                          :type 'veh-dat
                          :key 'sensor-1-cycle-time
                          :item 50))

(add-to-bb (MAKE-INSTANCE 'data-object          ;left-side flag
                          :level 'lo
                          :type 'sys-dat
                          :key 'left-side
                          :item T))

(add-to-bb (MAKE-INSTANCE 'data-object          ;right-side flag
                          :level 'lo
                          :type 'sys-dat
                          :key 'right-side
                          :item T))

(add-to-bb (MAKE-INSTANCE 'data-object          ;list of knowledge
                          :level 'hi            ;sources that exist
                          :type 'control-specs
                          :key 'emergency-ks-list
                          :item NIL))

(add-to-bb (MAKE-INSTANCE 'data-object          ;list of knowledge
                          :level 'hi            ;sources that exist
                          :type 'control-specs
                          :key 'notification-ks-list
                          :item (list 'found-left
                                      'found-right
                                      'missing-left
                                      'missing-right)))
```

```
(add-to-bb (MAKE-INSTANCE 'data-object          ;list of knowledge
                          :level 'hi            ;sources that exist
                          :type 'control-specs
                          :key 'available-ks-list
                          :item (LIST 'find-max-row
                                      'find-min-row
                                      'find-edges
                                      'find-left-side
                                      'find-right-side
                                      'dispatcher
                                      'purge-bb)))

(add-to-bb (make-instance 'data-object          ;default priority
                          :level 'hi            ;for data-driven
                          :type 'control-specs  ;KS-invocation
                          :key 'data-driven-priority
                          :item 6))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;******* KSDEF.LISP ********************************************
;******* Last-edit 05/28/87 ************************************

;;;This file, ksdef.lisp, contains flavor definitions and methods
; which define the knowledge sources of our blackboard architecture.
; The checker and actor flavors should be components of all KSs, but
; more specialized KSs may incorporate additional instance variables
; and may redefine the check and invoke methods as needed.  The KS
; flavor, KS-1, is the most basic type of KS, and is intended to model
; those KSs with specific image-derived input and output.
;;;


(DEFFLAVOR checker
        ( conditions)                          ;any additional cond'ns
        ()                                     ;no component flavors
  :INITABLE-INSTANCE-VARIABLES
  :WRITABLE-INSTANCE-VARIABLES                 ;possible controller changes
  (:REQUIRED-INSTANCE-VARIABLES
                type
                inv-disp-wind
                input-locator
                output-locator )
  :ABSTRACT-FLAVOR )                           ;non-instantiable


(DEFFLAVOR actor
        ( descriptor )                         ;holds invocation particulars
        ()                                     ;no component flavors
  :WRITABLE-INSTANCE-VARIABLES
  (:REQUIRED-INSTANCE-VARIABLES
                type
                act-disp-wind
                input-locator
                output-locator )
  :ABSTRACT-FLAVOR )                           ;non-instantiable


(DEFFLAVOR ks-1                                ;"base" ks flavor
        ( type                                 ;specifies actor function
          inv-disp-wind                        ;display window (bar graph)
          act-disp-wind                        ;display window (bar-graph)
          input-locator                        ;(level type) of input data
          output-locator )                     ;(level type) of output data
        ( checker                              ;checker and actor methods
          actor )
  :INITABLE-INSTANCE-VARIABLES
  :READABLE-INSTANCE-VARIABLES
  (:CONC-NAME ks- ))                           ;reference generically as ks


(SETQ find-min-row                             ;find-min-row KS template
      (MAKE-INSTANCE 'ks-1
                     :conditions NIL           ;no xtra conditions
                     :inv-disp-wind *mnri*
                     :act-disp-wind *mnra*
                     :type "find-min-row       ;find-min-row is KS function
                     :input-locator '((lo max-row) (lo image))
                     :output-locator '((lo min-row))))
```

```
(SETQ find-max-row                                  ;find-max-row KS template
      (MAKE-INSTANCE 'ks-1
                     :conditions NIL
                     :inv-disp-wind *mxri*
                     :act-disp-wind *mxra*
                     :type "find-max-row
                     :input-locator '((lo image))
                     :output-locator '((lo max-row))))

(SETQ dispatcher                                    ;dispatcher KS template
      (MAKE-INSTANCE 'ks-1
                     :conditions                    ;xtra condition - most recent
                     #'(LAMBDA (IGNORE IGNORE)
                                                    ;image must be new data
                         (LET* ((it (FIRST *image-queue2*))
                                (key (time-tag it)))
                           (NULL (GETHASH key
                                          (get-table '(lo image))))))
                     :inv-disp-wind *imgi*
                     :act-disp-wind *imga*
                     :type "dispatcher
                     :input-locator '()
                     :output-locator '((lo image))))


(SETQ find-edges                                    ;find-edges KS template
      (MAKE-INSTANCE 'ks-1
                     :conditions NIL
                     :inv-disp-wind *edgi*
                     :act-disp-wind *edga*
                     :type "find-edges
                     :input-locator '((lo min-row) (lo max-row) (lo image))
                     :output-locator '((lo left-edge) (lo right-edge))))


(SETQ find-left-side                                ;find-left-side KS template
      (MAKE-INSTANCE 'ks-1
                     :conditions                    ;xtra condition - must be
                     #'(LAMBDA (k il)               ;more that one edge point
                         (LET ((it (get-table
                                    (FIRST il))))
                           (> (FILL-POINTER (get-item
                                             (GETHASH k it)))
                              1)))
                     :inv-disp-wind *lsdi*
                     :act-disp-wind *lsda*
                     : type "find-left-side
                     :input-locator '((lo left-edge))
                     :output-locator '((lo left-side))))


(SETQ find-right-side                               ;find-right-side KS template
      (MAKE-INSTANCE 'ks-1
                     :conditions                    ;xtra condition - same as
                     #'(LAMBDA (k il)               ;for find-left-side
                         (LET ((it (get-table
                                    (FIRST il))))
                           (> (FILL-POINTER (get-item
                                             (GETHASH k it)))
                              1)))
                     :inv-disp-wind *rsdi*
                     :act-disp-wind *rsda*
                     :type "find-right-side
```

```
                    :input-locator '((lo right-edge))
                    :output-locator '((lo right-side))))



;;;*****************************************************************************
;;;
;;;missing-found-sides KSDEFS

(SETQ missing-left                              ;missing-left KS template
      (MAKE-INSTANCE 'ks-1
                    :conditions                 ;xtra conditions -
                    #'(LAMBDA (k il)            ;left-side flag T and
                        (LET ((it (get-table          ;must be less than
                                  (FIRST il))))       ;two edge points
                          (AND
                            (get-sys 'left-side)
                            (< (FILL-POINTER (get-item
                                             (GETHASH k it)))
                               2)
                            (< (create-time (gethash 'left-side
                                                     (get-table '(lo sys-dat))))
                               (create-time (gethash k it))))))

                    :inv-disp-wind *flgi*
                    :act-disp-wind *flga*
                    :type "missing-left
                    :input-locator '((lo left-edge))
                    :output-locator NIL))



(SETQ found-left                                ;found-left KS template
      (MAKE-INSTANCE 'ks-1
                    :conditions                 ;xtra conditions -
                    #'(LAMBDA (k il)            ;left-side flag NIL and
                        (LET ((it (get-table          ;must be more than
                                  (FIRST il))))       ;one edge point
                          (AND
                            (Null (get-sys 'left-side))
                            (> (FILL-POINTER (get-item
                                             (GETHASH k it)))
                               1)
                            (< (create-time (gethash 'left-side
                                                     (get-table '(lo sys-dat))))
                               (create-time (gethash k it))))))

                    :inv-disp-wind *flgi*
                    :act-disp-wind *flga*
                    :type "found-left
                    :input-locator '((lo left-edge))
                    :output-locator NIL))

(SETQ missing-right                             ;missing-right KS template
      (MAKE-INSTANCE 'ks-1
                    :conditions                 ;xtra conditions -
```

```
                    #'(LAMBDA (k il)              ;right-side flag T and
                        (LET ((it (get-table          ;must be less than
                                  (FIRST il))))       ;two edge points
                          (AND
                            (get-sys 'right-side)
                            (< (FILL-POINTER (get-item
                                             (GETHASH k it)))
                               2)
                            (< (create-time (gethash 'right-side
                                                     (get-table '(lo sys-dat))))
                               (create-time (gethash k it))))))

                    :inv-disp-wind *flgi*
                    :act-disp-wind *flga*
                    :type "missing-right
                    :input-locator '((lo right-edge))
                    :output-locator NIL))



(SETQ found-right                                 ;found-right KS template
      (MAKE-INSTANCE 'ks-1
                     :conditions                  ;xtra conditions -
                     #'(LAMBDA (k il)             ;right-side flag NIL and
                         (LET ((it (get-table          ;must be more than
                                   (FIRST il))))       ;one edge point
                           (AND
                             (Null (get-sys 'right-side))
                             (> (FILL-POINTER (get-item
                                              (GETHASH k it)))
                                1)
                             (< (create-time (gethash 'right-side
                                                      (get-table '(lo sys-dat))))
                                (create-time (gethash k it))))))

                     :inv-disp-wind *flgi*
                     :act-disp-wind *flga*
                     :type "found-right
                     :input-locator '((lo right-edge))
                     :output-locator NIL))

(setq purge-bb
      (make-instance 'ks-1
                     :conditions
                     #'(Lambda (&rest ignore)
                         (let* ((it (get-table '(lo image)))
                                (tmp 0))
                           (loop for x being the hash-elements of it
                                 doing
                             (setq tmp (1+ tmp)))
                           (> tmp 25)))
                     :inv-disp-wind *gci*
                     :act-disp-wind *gca*
                     :type "purge-bb
                     :input-locator NIL
                     :output-locator NIL))



(SETQ missing-both                                ;missing-both KS template
```

```
(MAKE-INSTANCE 'ks-1
                :conditions                    ;xtra conditions -
                #'(LAMBDA (IGNORE IGNORE)
                                               ;both left-side & right-side
                                               ;flags must be NIL
                    (AND
                      (NULL (get-sys 'right-side))
                      (NULL (get-sys 'left-side))))
                :inv-disp-wind *flgi*
                :act-disp-wind *flga*
                :type "missing-both
                :input-locator NIL
                :output-locator NIL))

(SETQ check-clear                              ;check-clear KS template
    (MAKE-INSTANCE 'ks-1
                :conditions                    ;xtra conditions -
                #'(LAMBDA (IGNORE IGNORE)
                                               ;both left-side & right-side
                                               ;flags must be NIL
                    (AND
                      (NULL (get-sys 'right-side))
                      (NULL (get-sys 'left-side))))
                :inv-disp-wind *clri*
                :act-disp-wind *clra*
                :type "check-clear
                :input-locator '((lo min-row) (lo max-row) (lo image))
                :output-locator NIL))
```

```
; -*- Mode:  lisp; Base: 10; Package: CL-USER; Syntax: Common-Lisp -*-

;******** KSFUNS.LISP ************************************************
;******** Last-edit 05/29/87 *****************************************

;;;-----straight from Mort's code---------

;;; +=
; A macro for incrementing numerical components of generalized variables.
; Can be used with any lisp object accessible with setf.
;;;

(DEFMACRO += (num-slctr num)
  '(SETF ,num-slctr (+ ,num-slctr ,num)))

;;; LR-DATA
; A structure for accumulating linear regression data.
;;;

(DEFSTRUCT (lr-data
             (:CONC-NAME NIL))
  (n 0)
  (sumx 0.0)                                    ; sum over i of x[i]
  (sumy 0.0)                                    ; sum over i of y[i]
  (sumx2 0.0)                                   ; sum over i of (x[i] - sumx / n) ^ 2
  (sumxy 0.0))                                  ; sum over i of
                                                ; (x[i] - sumx / n) * (y[i] - sumy / n)

;;; ADD-POINT
; Add an observation to the set of data points being accumulated.
;;;

(DEFUN add-point (lr x y)
"ARGUMENTS: lr is type lr-data, x and y are numbers"
  (+= (n lr) 1)
  (+= (sumx lr) x)
  (+= (sumy lr) y)
  (WHEN (> (n lr) 1)
    (LET* ((n (n lr))
           (dx (- x (/ (sumx lr) n)))
           (dy (- y (/ (sumy lr) n))))
      (+= (sumx2 lr) (/ (* n (*  dx dx)) (1- n)))
      (+= (sumxy lr) (/ (* n (*  dx dy)) (1- n))))))

;;; LR-COEFS
; Compute the coefficients a, b of the linear regression line y = ax + b
; and return them as a list.
;;;

"ARGUMENT: lr is type lr-data"
(DEFUN lr-coefs (lr)
  (LET* ((n (n lr))
         (mu-x (/ (sumx lr) n))
         (mu-y (/ (sumy lr) n))
         (a (/ (sumxy lr) (sumx2 lr)))
         (b (- mu-y (* a mu-x))))
    (LIST a b)))

;;; LR-EVAL
; Given a list of form returned be lr-coeffs and a value of x, the independent
; variable, return the value of y, the dependent variable.
;;;
```

```
(DEFUN lr-eval (coeff-list x)
"ARGUMENTS: coeff-list is a list of two numbers, x is a number"
  (+ (* (FIRST coeff-list) x) (SECOND coeff-list)))

;;;---------end of Mort's direct code-------


;;; GET-VEHICLE
; This function retrieves data about the vehicle from a table in the
; blackboard.  Key is a descriptive key for the element of the table
; needed.
;;;

(DEFUN get-vehicle (key)
  (get-item
    (GETHASH key
             (get-table '(lo veh-dat )))))

;;; FIND-MAX-ROW
; This function is part of the FIND-MAX-ROW KS and is used to delimit the
; usable portion of the image by finding the topmost usable row.  This is
; done by searching down the image until a row with something other than
; zero-valued pixels is found.  Only the middle (vehicle-width) of the
; image is checked.
;;;

(DEFUN find-max-row (curr-img key)
  (DECLARE (SPECIAL *mxra*))
  (LET* ((row-size (row-size curr-img ))
         (max-row (- row-size  1 ))                    ;begin with max-row of 127
         (black-pixels 0 )                             ;init counter
         (half-width (get-vehicle 'half-width))
         (center-col (center-col curr-img))
         (width (+ half-width half-width ))
         (proc-id (LIST "find-max-row key)))
    (LOOP FOR row FROM 0 TO (- row-size  1)            ;search each row
          DO
      (SETQ black-pixels                               ;get number of 'zero' pixels
            (LOOP FOR col
                      FROM (+ (- center-col half-width) 1 )        ;search each column
                      TO (+ center-col half-width )                ;in center width
                  AND FOR px FIRST (AREF (image curr-img ) row col )
                      THEN (AREF (image curr-img ) row col ) ;
                  COUNT ( = px 0 ) INTO n                  ;count the zeroes
                  FINALLY (RETURN n )))

      (WHEN ( < black-pixels width ) (SETQ max-row row )))         ;found the max-row
    (add-to-bb
      (MAKE-INSTANCE 'data-object                      ;make and return a max-row object
                     :item max-row                     ;item is max-row
                     :level 'lo
                     :type 'max-row
                     :key key ))                       ;id is the image's time-tag
    (rem-from-bb (GETHASH proc-id                      ;clear the element from have-done
                          (get-table '(hi have-done))))
    (decr-graph *mxra*)))                              ;update the active KS bar graph


;;; FIND-MIN-ROW
; This function is part of the FIND-MIN-ROW KS.  It is used to delimit the usable
; portion of the image with respect to processing by finding the minimun row of
; usable data.  This is done by processing each row from the maximum usable row
; down to the bottom.  The row above the first row found which contains pixels
```

```
; greater than 250 (marked) or equal to zero within a width of the vehicle about
; the center of the image, is returned as the min-row.
;;;

(DEFUN find-min-row (max-row curr-img key)
  (DECLARE (SPECIAL *mnra*))
  (LET* ((min-row 0)
         (grey-pixels 0)
         (half-width (get-vehicle 'half-width))
         (center-col (center-col curr-img))
         (width (+ half-width half-width))
         (proc-id (LIST "find-min-row key)))
    (LOOP FOR row FROM (- max-row 1) DOWNTO 0
          DO
      (SETQ grey-pixels
            (LOOP FOR col FROM ( + ( - center-col half-width) 1 )
                               TO ( + center-col half-width)
                  AND FOR px FIRST (AREF (image curr-img) row col )
                             THEN  (AREF (image curr-img) row col )
                  COUNT ( AND ( < px 250 ) ( > px 0 )) INTO n
                  FINALLY ( RETURN n )))
      (WHEN ( < grey-pixels width )
        ( SETQ min-row (+ row 1 ))
        ( LOOP-FINISH )))
    (add-to-bb (MAKE-INSTANCE 'data-object      ;add the min-row to the blackboard
                              :level 'lo
                              :type 'min-row
                              :item min-row
                              :key key))
  (rem-from-bb (GETHASH proc-id                 ;clear the KS from have-done
                        (get-table '(hi have-done))))
  (decr-graph *mnra*)))                         ;update the active KS bar graph


;;;FIND-EDGES
; This function is part of the FIND-EDGES KS and produces to data items:
; left-edge and right-edge. The data items are arrays of row-column points
; which are found marked in the image. The image is only scanned between
; maxrow and minrow, excluding the width of the vehicle about the center column.
; Only the first point found while scanning out from the center is saved in
; the array.
;;;

(DEFUN find-edges ( min-row max-row curr-img key)
  (DECLARE (SPECIAL *edga*))
  (LET* ((edge-color 254)
         (row-size (row-size curr-img))
         (col-size (col-size curr-img))
         (left-of-center
           (MAKE-ARRAY row-size :FILL-POINTER 0))   ;make the data arrays
         (right-of-center
           (MAKE-ARRAY row-size :FILL-POINTER 0))
         (center-col (center-col curr-img))
         (half-width (get-vehicle 'half-width))
         (proc-id (LIST "find-edges key)))
    (LOOP FOR row FROM (+ min-row 1) TO (- max-row 1)
                                                  ;cover usable image
          DO
      (LET ((path-lf (- center-col half-width))   ;avoid processing center
            (path-rt (+ center-col half-width))
            (max-col (- col-size 1)))
        (LOOP FOR col FROM path-lf DOWNTO 0 ;scan left in the row
              WHEN (= (AREF (image curr-img) row col)
```

```
                    edge-color)
                DO
                  (VECTOR-PUSH (LIST row col)
                               left-of-center)
                  (LOOP-FINISH))
        (LOOP FOR col FROM path-rt TO max-col ;scan right in the row
              WHEN (= (AREF (image curr-img) row col)
                    edge-color)
                DO
                  (VECTOR-PUSH (LIST row col)
                               right-of-center )
                  (LOOP-FINISH))))
    (add-to-bb (MAKE-INSTANCE 'data-object      ;add the data to the blackboard
                              :item left-of-center
                              :type 'left-edge
                              :level 'lo
                              :key  key ))
    (add-to-bb (MAKE-INSTANCE 'data-object
                              :item right-of-center
                              :type 'right-edge
                              :level 'lo
                              :key  key ))
    (rem-from-bb (GETHASH proc-id
                          (get-table '(hi have-done))))
    (decr-graph *edga*)))


;;;FIND-LEFT-SIDE
; This is the function of the FIND-LEFT-SIDE KS.  The left-edge data points are
; processed to find a best fit line.  The data written to the blackboard from
; this function consist of lists of the slope and intercept values of the best
; fit line.
;;;

(DEFUN find-left-side (edge key)
  (DECLARE (SPECIAL *lsda*))
  (LET ((left-data (make-lr-data))
        (proc-id (LIST "find-left-side key)))
    (LOOP FOR indx FROM 0 TO (- (FILL-POINTER edge) 1)
          DO
      (LET ((datum (AREF edge indx)))
        (add-point left-data (FIRST datum) (SECOND datum))))
    (add-to-bb (MAKE-INSTANCE 'data-object
                              :item (lr-coefs left-data)
                              :type 'left-side
                              :level 'lo
                              :key key))
  (rem-from-bb (GETHASH proc-id
                        (get-table '(hi have-done))))
  (decr-graph *lsda*)))


;;;FIND-RIGHT-SIDE
; This is the function of the FIND-RIGHT-SIDE KS.  It produces the same thing
; as the FIND-LEFT-SIDE function above, only it operates on the right-edge
; data points.
;;;

(DEFUN find-right-side (edge key)
  (DECLARE (SPECIAL *rsda*))
  (LET ((right-data (make-lr-data))
        (proc-id (LIST "find-right-side key)))
    (LOOP FOR indx FROM 0 TO (- (FILL-POINTER edge) 1)
```

```
        DO
      (LET ((datum (AREF edge indx)))
        (add-point right-data (FIRST datum) (SECOND datum))))
    (add-to-bb (MAKE-INSTANCE 'data-object
                              :item (lr-coefs right-data)
                              :type 'right-side
                              :level 'lo
                              :key key))
  (rem-from-bb (GETHASH proc-id
                        (get-table '(hi have-done))))
  (decr-graph *rsda*)))


;;; DISPATCH
; This funtion is used by the DISPATCHER KS.  It adds a given image object
; to the blackboard.
;;;

(DEFUN dispatch (img key)
  (add-to-bb (MAKE-INSTANCE 'data-object
                            :item img
                            :key key
                            :level 'lo
                            :type 'image)))


;;;DISPATCHER
; This is the function for the DISPATCHER KS.  It has several unusual conditions
; built in.  If no key is given to this function, it checks if the most recent
; image on the queue is new data with respect to the blackboard.  If it is, it
; fetches it.  If a key was passed, but it reflects an image not yet received from
; the sensor, this function waits for the desired image and then fetches it.  If it
; has been received from the sensor, this function searches the queue for the image
; and returns it if it is found.
;;;

(DEFUN dispatcher (&OPTIONAL (key 0))
  (DECLARE (SPECIAL *image-queue2* *imga*))
  (LET ((proc-id (LIST "dispatcher key)))
    (IF (EQUALP key 0)
        (LET ((curr-img (FIRST *image-queue2*)))
          (IF (NULL (GETHASH (time-tag curr-img)
                             (get-table '(lo image))))
              (dispatch curr-img (time-tag curr-img))))      ;image is new data
        (LET ((curr-img (FIRST *image-queue2*)))
          (IF (> key (time-tag curr-img))                    ;image not in queue yet
              (LOOP UNTIL
                    (EQUALP (time-tag (FIRST *image-queue2*))
                            key)
                FINALLY (dispatch (FIRST *image-queue2*)
                                  (time-tag (first *image-queue2*))))
              (LOOP FOR img IN *image-queue2*
                    DOING
                (IF (EQUALP (time-tag img) key)              ;found desired image
                    (dispatch img key))))))
    (rem-from-bb (GETHASH proc-id
                          (get-table '(hi have-done))))
    (decr-graph *imga*)))
```

```
;;;*****************************************************************************
;;;
; Functions for missing-found-sides

;;;*****************************************************************************
;;;
;
;;; GET-SYS
; This function retrieves data about the system from a table in the
; blackboard.  Key is a descriptive key for the element of the table
; needed.
;;;
;;;

(DEFUN get-sys (key)
  (get-item
    (GETHASH key
             (get-table '(lo sys-dat )))))


(DEFUN missing-left (IGNORE key)
  (DECLARE (SPECIAL *flga* *flag-window*
                    *demo-window-alu* *danger-color*))
  (LET ((proc-id (LIST "missing-left key))
        (tmp-alu *demo-window-alu*))
    (SETF                                          ;(get-sys 'left-side)
                                                              ;why doesn't this work??
      (get-item (GETHASH 'left-side (get-table '(lo sys-dat))))      ;and this does??
      NIL)
    (setf (create-time
            (gethash 'left-side
                     (get-table '(lo sys-dat)))) (zl:time))
    (SEND tmp-alu :SET-FILL-DATA *danger-color*)
    (flag-condition *flag-window*
                    "CAUTION!  Left side of road not detected!"
                    (left *flag-window*)
                    (left-missing *flag-window*)
                    tmp-alu)
    (decr-graph *flga*)
    (rem-from-bb (GETHASH proc-id
                          (get-table '(hi have-done))))))


(DEFUN found-left (Ignore key)
  (DECLARE (SPECIAL *flga* *flag-window*))
  (SETF                          ;(get-sys 'left-side)        ;why doesn't this work??
    (get-item (GETHASH 'left-side (get-table '(lo sys-dat))))          ;and this does??
    T)
  (SETF (create-time
          (GETHASH 'left-side
                   (get-table '(lo sys-dat)))) (ZL:TIME))
  (erase-condition *flag-window* (left-missing *flag-window*))
  (LET ((proc-id (LIST "found-left key)))
    (rem-from-bb (GETHASH proc-id
                          (get-table '(hi have-done)))))
  (decr-graph *flga*))


(DEFUN missing-right (IGNORE key)
  (declare (special *flga* *flag-window*
                    *demo-window-alu* *danger-color*))
  (setf (get-item
          (gethash 'right-side
                   (get-table '(lo sys-dat)))) NIL)
  (setf (create-time
          (gethash 'right-side
```

```
                    (get-table '(lo sys-dat)))) (zl:time))
  (let ((tmp-alu *demo-window-alu*)
        (proc-id (list "missing-right key)))
    (send tmp-alu :set-fill-data *danger-color*)
    (flag-condition *flag-window*
                    "CAUTION!  Right side of road not detected!"
                    (left *flag-window*)
                    (right-missing *flag-window*)
                    tmp-alu)
    (rem-from-bb (gethash proc-id (get-table '(hi have-done))))
    (decr-graph *flga*)))


(DEFUN found-right (IGNORE key)
  (declare (special *flga* *flag-window*))
  (setf   (get-item
            (gethash 'right-side
                     (get-table '(lo sys-dat)))) t)
  (setf (create-time
            (gethash 'right-side
                     (get-table '(lo sys-dat)))) (zl:time))
  (erase-condition *flag-window* (right-missing *flag-window*))
  (let ((proc-id (list "found-right key)))
    (rem-from-bb (gethash proc-id (get-table '(hi have-done)))))
  (decr-graph *flga*))


(defun purge-bb (key)
  (declare (special *gca* *image-queue1*))
  (let* ((now (time-tag (first *image-queue1*)))
         (lo-limit (- now (* 10 (get-vehicle 'sensor-1-cycle-time))))
         (image-table (get-table '(lo image)))
         (max-row-table (get-table '(lo max-row)))
         (min-row-table (get-table '(lo min-row)))
         (left-edge-table (get-table '(lo left-edge)))
         (right-edge-table (get-table '(lo right-edge)))
         (left-side-table (get-table '(lo left-side)))
         (right-side-table (get-table '(lo right-side)))
         (table-list
           (list image-table max-row-table min-row-table
                 left-edge-table right-edge-table
                 left-side-table right-side-table)))
    (loop for table in table-list
          doing
      (loop for x being the hash-elements of table
            with-key k
            doing
        (if (< k lo-limit)
            (remhash k table)))))
  (let ((proc-id (list "purge-bb key)))
    (rem-from-bb (gethash proc-id (get-table '(hi have-done)))))
  (decr-graph *gca*))


;;Function for KS 'missing-both which sounds an alarm.
(DEFUN missing-both (key)
```

```
  (declare (special *flga*))
  (let ((proc-id (list "missing-both key)))
    (zl-user:%beep-ignoring-most-issues 500 125000)          ;sound alarm
    (zl-user:%beep-ignoring-most-issues 100 125000)
    (rem-from-bb (gethash proc-id (get-table '(hi have-done))))
    (decr-graph *flga*)))


(DEFUN check-clear ( min-row max-row curr-img key)
  (DECLARE (SPECIAL *clra*))
  (LET* ((edge-color 254)
         (obstaclep NIL)
         (center-col (center-col curr-img))
         (half-width (get-vehicle 'half-width))
         (proc-id (LIST "check-clear key)))
    (LOOP FOR row FROM (+ min-row 1) TO (- max-row 1)
                                                    ;cover usable image
          DO
      (LET ((path-lf (- center-col half-width))     ;process center
            (path-rt (+ center-col half-width)))
        (LOOP FOR col FROM path-rt DOWNTO path-lf   ;scan left in the row
              WHEN (= (AREF (image curr-img) row col)
                      edge-color)
                DO
                  (SETQ obstaclep T)
                  (LOOP-FINISH))))
    (IF (NOT obstaclep)                             ;clear path detected
            NIL
            (loop repeat 5 doing (beep)))

    (rem-from-bb (GETHASH proc-id
                          (get-table '(hi have-done))))
    (decr-graph *clra*)))


;;; This is taken directly from rel-7>examples>audio-examples.lisp.  One
;;; change was made to account for the difference between zetalisp syntax
;;; and common-lisp; the function // was changed to zl:/.  In Zetalisp
;;; the / is the quoting character so it must be doubled.  We are using
;;; common-lisp here so it remains zl:/.
;;;
;;; This is roughly the LM-2 %BEEP equivalent, but there are many
;;; reasons why this version can fail as the default.  It is basically a
;;; modified SQUARE-WAVE.
;
(defun %beep-ignoring-most-issues (frequency duration)
  (audio:with-audio ()
    (let* ((start (audio:audio-index))
           (samples-per-cycle (sys:round audio:*sample-rate* frequency))
           (samples-first-half (zl:/ samples-per-cycle 2))  ;changed // to zl:/
           (samples-second-half (- samples-per-cycle samples-first-half)))
      ;; can't nest loops, so we have to do the outer loop with a jump
      ;; and bash the location when time has elapsed.
      (audio:audio-loop (samples-first-half)
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample 0.9))))
      (audio:audio-loop (samples-second-half)
        (audio:computing-immediate-audio-samples (1)
          (audio:push-immediate-audio-sample (audio:float-sample -0.9))))
      ;; This is the tricky part.  We need to put a jump to the
      ;; beginning, but we need to know where it is so we can cause it
```

```
;; to fall through.  We also need a flag so we know when the audio
;; has stopped so we can exit.  If we simply exited without
;; waiting, the WITH-AUDIO form could turn off the sound prematurely.
(let* (;; get the index that we will eventually bash and put in a
       ;; jump back to the start.
       (jump-index (prog1 (audio:audio-index) (audio:push-audio-jump start)))
       ;; reserve (and reset) an audio flag.
       (flag-index (audio:reserve-audio-flags 1))
       ;; reserve-audio-flags puts in a jump command around the
       ;; flags it reserves, so we could have gotten the
       ;; fall-through index after pushing the jump command.
       ;; Anyway, get the index of the fall-through location.
       (fall-through-index (audio:audio-index)))
  ;; when we bash the jump command the microcode will jump to here
  ;; instead, which will cause the flag to get zeroed and the
  ;; audio facility to stop.  Both events happen atomically as far
  ;; as Lisp can tell because no samples are output in the
  ;; intervening time.
  (audio:push-audio-zero-flag flag-index)
  (audio:push-audio-stop)
  ;; fire it up!
  (audio:audio-start start)
  ;; wait the appropriate number of microseconds.
  (loop with start-time = (sys:%microsecond-clock)
        until ( (zl-user:%32-bit-difference (sys:%microsecond-clock) start-time) duration))
  ;; Here is where we bash the argument of the jump command to
  ;; instead jump to the fall-through code.
  (audio:modify-audio-command-arg fall-through-index :index jump-index)
  ;; Wait for the microcode to get to the flag and stop before we exit.
  (audio:wait-for-audio-flag flag-index "%BEEP")))))
```

```
;; to fall through.  We also need a flag so we know when the audio
;; has stopped so we can exit.  If we simply exited without
;; waiting, the WITH-AUDIO form could turn off the sound prematurely.
(let* (;; get the index that we will eventually bash and put in a
       ;; jump back to the start.
       (jump-index (prog1 (audio:audio-index) (audio:push-audio-jump start)))
       ;; reserve (and reset) an audio flag.
       (flag-index (audio:reserve-audio-flags 1))
       ;; reserve-audio-flags puts in a jump command around the
       ;; flags it reserves, so we could have gotten the
       ;; fall-through index after pushing the jump command.
       ;; Anyway, get the index of the fall-through location.
       (fall-through-index (audio:audio-index)))
  ;; when we bash the jump command the microcode will jump to here
  ;; instead, which will cause the flag to get zeroed and the
  ;; audio facility to stop.  Both events happen atomically as far
  ;; as Lisp can tell because no samples are output in the
  ;; intervening time.
  (audio:push-audio-zero-flag flag-index)
  (audio:push-audio-stop)
  ;; fire it up!
  (audio:audio-start start)
  ;; wait the appropriate number of microseconds.
  (loop with start-time = (sys:%microsecond-clock)
        until ( (zl-user:%32-bit-difference (sys:%microsecond-clock) start-time) duration))
  ;; Here is where we bash the argument of the jump command to
  ;; instead jump to the fall-through code.
  (audio:modify-audio-command-arg fall-through-index :index jump-index)
  ;; Wait for the microcode to get to the flag and stop before we exit.
  (audio:wait-for-audio-flag flag-index "%BEEP")))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-


;;;This file, checker.lisp, contains the methods which perform the checking
; operations of the knowledge sources on themselves.  A documentation blurb
; precedes each method.


;;;MATCH-REQUEST-TIME-1
;
;;; This method finds, if possible; an input data item which matches the
; required item, in the time frame variable, within one sensor-1 cycle.
; If found, the correct time (key) is returned to the checker.
;
(DEFMETHOD (match-request-time-1 checker) (key)
      (LET ((request-time key)
            (turn-around (get-vehicle 'sensor-1-cycle-time)))
        (multiple-value-bind (times ignore)
            (round request-time turn-around )
          (* times turn-around))))


;;;EXISTSP
;
;;;This method checks for existance of any secondary data necessary for the
; operation of a KS.  It is defined as a method on the KS-1 flavor because
; the actor also makes use of it.
;
(DEFMETHOD (existsp ks-1) (key)
  (IF ( NULL input-locator)
      T                                          ;return t if no other inputs
      (LOOP FOR table-spec IN  input-locator
            ALWAYS (GETHASH key                  ;return t if all inputs exist
                            (get-table table-spec)))))


;;;CHECK-DATA-TABLE
;
;;;This method performs several checks on a KS, namely :  it checks if a
; description matching this KS is on the marked table (already invoked);
; it checks whether or not the data that would be produced by this KS
; already exists, and it checks any additional conditions which may be
; contained in the KS.  If the result doesn't exist, an equivalent KS is
; not already processing, and if all internal conditions check true, then
; this method checks for existance of any additional input data.  If all
; other required data exists, this method creates an invocable-ks-object
; and places it on the blackboard.
; Note that this method is utilized for every element of data in the table
; located by the first element of the input-locator, and that this method
; produces only data-driven invocables.
;
(DEFMETHOD ( check-data-table checker ) ( key )
  (let ((marked-table (get-table '(hi have-done)))
        (marker (LIST type key)))
    (IF (AND (OR (NULL output-locator)
                 (NULL (GETHASH
                         key
                         (get-table
                           (first output-locator)))))          ;result doesn't exist
             (NULL (GETHASH marker marked-table))              ;not already processing
             (OR (NULL conditions)
                 (APPLY conditions
                        (LIST key input-locator))))            ;all conditions satisfied
```

```
        (IF (existsp SELF key)
            (PROGN

              (add-to-bb
                (MAKE-INSTANCE 'ks-description
                               :ks-ref       type
                               :input-key  key
                               :requestor  NIL
                               :priority     NIL
                               :level        'hi
                               :type         'invocable-ks-table
                               :key          (LIST type key)
                               :data         T ) )
              (incr-graph inv-disp-wind))))))


;;;CHECK-REQUEST-TABLE
;
;;;This method is utilized upon each element of the request table, for each
; KS checked.  The method checks if the type of data requested is that which
; is produced by the current KS.  If not, nothing is done.  If it is, then
; the method attempts to located a more exact time-frame reference (key) by
; use of the MATCH-REQUEST-TIME-1 method.  If it cannot be matched, or if
; some data for the time-frame are missing, the KS-descrip is marked to
; indicate this.  The KS description is written to the invocable-ks-table of
; the blackboard.  Priority and requestor information are included in the
; description.
;
(DEFMETHOD ( check-request-table checker ) ( key value)
  (loop for locator in output-locator
        doing
    (LET (( output-type ( SECOND locator))
          ( input-key )
          ( existing-data ))
      (IF ( NOT ( EQUALP output-type
                  ( request-req-type value )))
          NIL
          (SETQ input-key
                (match-request-time-1
                  SELF
                  (request-input-key value)))
          (IF input-key
              (SETF (request-input-key value) input-key))
          (IF (OR (NULL input-key)
                  (null (existsp SELF input-key)))
              (SETQ existing-data NIL)
              (SETQ existing-data T))
          (let ((marked-table (get-table '(hi have-done)))
                (marker (list type (second key))))
            (if (or (gethash marker marked-table)
                    (gethash (request-input-key value)
                             (get-table locator)))
                NIL
                (add-to-bb
```

```
                (MAKE-INSTANCE 'KS-description
                                :ks-ref  type
                                :input-key (request-input-key value)
                                :requestor (request-requestor value)
                                :priority (request-priority value)
                                :level 'hi
                                :type 'invocable-ks-table
                                :key (LIST type (request-input-key
                                                  value))
                                :data existing-data ))
            (incr-graph inv-disp-wind )))))))


;;;CHECK
;
;;;This is the primary checking method; checking a KS can be done
; merely by saying (check KS).  This method obtains the primary input
; table for the KS, and loops through the table, applying CHECK-DATA-TABLE
; to each element.  It then loops through the request table, applying
; CHECK-REQUEST-TABLE to each element.
;
(DEFMETHOD (check checker) ()
  (LET ((input-table)
        (request-table (get-table '( hi requests ))))
    (IF (null input-locator)
        (let ((key (time-tag (first *image-queue1*))))
          (check-data-table self key ))
        (setq input-table
              (get-table (first input-locator)))

        (LOOP FOR value BEING THE HASH-ELEMENTS OF input-table
                        WITH-KEY key
              DO
          (check-data-table self key)))

    (LOOP FOR value BEING THE HASH-ELEMENTS OF request-table
                    WITH-KEY key
          do
      (check-request-table self key value))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;;;This file, actor.lisp, contains the methods used to invoke a knowledge
; source.  Detailed commentation for each method can be found directly prior
; to the code for that method.


;;;LIST-INPUTS
;
;;;This method should only be used when it is known that all required
; input data for a given knowledge source and time (key) exist.  This
; method locates each required datum and places it in a list of data.
; After including each item specified by the KS's input-locator list,
; this method appends the key to the data list, and returns the list.
;
(DEFMETHOD (list-inputs actor) (key)
  (IF ( NULL input-locator)
      (list key)                                          ;no inputs
      (LOOP FOR table-spec IN input-locator
            COLLECT (get-item
                      (GETHASH key
                               (get-table table-spec)))
              INTO input-list                             ;get every datum
            FINALLY (return (append input-list            ;append key and
                                    (list key))))))       ;return data list


;;;ACT-ON
;
;;;This method invokes a KS once all input data is known to exist.  A
; list of the data is obtained via the method LIST-INPUTS.  The function
; associated with the KS is run within a process, at the moment, with
; system default priority and quantum values.  The process is given a
; name composed of the KS-type and the input-key used to locate the data.
;
(DEFMETHOD (act-on actor) (ks-descrip)
  (LET* ((key (input-key ks-descrip))
         (input-data (list-inputs SELF key))              ;list the inputs
         (comm (APPEND (LIST 'PROCESS-RUN-FUNCTION
                                                          ;form command list
                             (FORMAT NIL "~a ~a "
                                     (EVAL type)
                                     key)
                             type)
                       input-data)))
    (EVAL comm)))                                         ;invoke the KS


;;;MAKE-REQUEST
;
;;;This method adds a request for a datum to the blackboard.  The requestor
; passed as a parameter is a KS-desription (i.e. an invokable KS).  The
; request is given the priority, the type, and the input-key of the requestor.
; The data-type and input-key are required to uniquely identify the info
; being requested.  The key of the new blackboard object is composed of the
; type of data requested and the key for that data.
;
(DEFMETHOD (make-request actor) (requestor data-type)
  (add-to-bb
    (MAKE-INSTANCE 'request                               ;make request object
                   :priority  (priority requestor)        ;pass on priority
```

```
                :quantity  NIL
                :req-type  data-type                    ;type of data requested
                :requestor (ks-ref requestor)           ;type of KS requesting
                :input-key (input-key requestor)        ;unique identifier
                :level     'hi
                :type      'requests
                :key       (LIST data-type
                                 (input-key requestor)))))


;;;FOO
;
;;;This method accepts as parameters a KS-description and a process ID.
; The ID is that of the process that placed requests for the input data
; required for invocation of the KS, but found non-existant. This method
; loops until all the required input data exist. Then it invokes the KS
; by means of the ACT-ON method, and kills the data-requesting process.
;
(DEFMETHOD (foo actor) (requestor process-id)
  (LOOP UNTIL
        (existsp SELF (input-key requestor)))
  (act-on SELF requestor)
  (SI:PROCESS-KILL process-id))


;;;MAKE-REQUESTS
;
;;;This method loops through the input-locator list of a KS-description,
; and checks if the data (identitied by the input key of the KS-descrip)
; exists. If it does, nothing is done; if it doesn't, the MAKE-REQUEST
; method is used to create a request on the blackboard for the missing
; data.
;
(DEFMETHOD (make-requests actor) (requestor)
  (LOOP FOR locator IN input-locator
        DO
    (IF (GETHASH (input-key requestor) (get-table locator))
        NIL
        (make-request SELF requestor (SECOND locator)))))


;;;INVOKE
;
;;;This is the primary actor method. This should be the method utilized
; by the controller to invoke a KS.
; This method first places a marker on the blackboard to indicate that
; the particular KS has been invoked. (The checker will insure that an
; equivalent KS-description is no longer invocable.) The KS function will
; remove the marker after the data resulting from the invocation have been
; written to the blackboard.
; This method then checks to see if the KS-descrip was the result of a
; request. If so, it removes the request from the blackboard.
; Next, the existance of the input data is checked. If all required input
; data exist, the KS is invoked via the ACT-ON method. If some data are
; known to be missing, MAKE-REQUESTS is utilized to request those data, and
; a process is created whose sole purpose is to execute the FOO method,
; i.e. to wait for the data and then to invoke the KS.
;
(DEFMETHOD (invoke actor) (ks-descrip)
  (LET ((marker (LIST type
                      (input-key ks-descrip))))
    (add-to-bb
      (MAKE-INSTANCE 'data-object               ;mark as acted on, to
```

```
                    :level 'hi                          ;avoid re-invocation
                    :type 'have-done
                    :key marker)
    (requestor ks-descrip)))
(incr-graph act-disp-wind)                              ;***display

(IF (requestor ks-descrip)                              ;remove request if exists
    (LET* ((req-table
              (get-table '(hi requests))))
      (loop for locator in output-locator
            with key
            with req
            doing
        (setq key
              (LIST (SECOND locator)
                    (input-key ks-descrip)))
        (setq req (GETHASH key req-table))
        (if req
            (rem-from-bb req)))))

(IF (data ks-descrip)                                   ;if data exists, just act
    (act-on SELF ks-descrip)
    (make-requests SELF ks-descrip)                     ;otherwise, request data
                                                        ;and make a wait process
    (LET ((process-name
            (STRING-APPEND (EVAL type)
                           (FORMAT NIL "~a"
                                   (input-key ks-descrip)))))
      (SETQ process-name (MAKE-PROCESS process-name))
      (PROCESS-PRESET process-name 'foo SELF ks-descrip process-name)
      (PROCESS-RESET-AND-ENABLE process-name))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

(defun check-emergency-cases ()
  (let ((ks-list (get-item
                   (gethash 'emergency-ks-list
                            (get-table '(hi control-specs )
                                       )))))
    (loop for ks in ks-list
          do
      (check (eval ks)))))

(defun check-flags-and-monitors ()
  (let ((ks-list (get-item
                   (gethash 'notification-ks-list
                            (get-table '(hi control-specs )
                                       )))))
    (loop for ks in ks-list
          do
      (check (eval ks)))))

(defun check-every-other-ks ()
  (let ((ks-list (get-item
                   (gethash 'available-ks-list
                            (get-table '(hi control-specs )
                                       )))))
    (loop for ks in ks-list
          do
      (check (eval ks)))))


(defun pick-a-ks ()
  (let ((invocables (get-table '(invocable-ks-table hi ))))
    (loop for poss-ks being the hash-elements of invocables
          with-key key
          do
      (princ key))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

(defun check-emergency-cases ()
  (let ((ks-list (get-item
                   (gethash 'emergency-ks-list
                     (get-table '(hi control-specs )
                       )))))
    (loop for ks in ks-list
          do
      (check (eval ks)))))

(defun check-flags-and-monitors ()
  (let ((ks-list (get-item
                   (gethash 'notification-ks-list
                     (get-table '(hi control-specs )
                       )))))
    (loop for ks in ks-list
          do
      (check (eval ks)))))

(defun check-every-other-ks ()
  (let ((ks-list (get-item
                   (gethash 'available-ks-list
                     (get-table '(hi control-specs )
                       )))))
    (loop for ks in ks-list
          do
      (check (eval ks)))))


(defun pick-a-ks ()
  (let ((invocables (get-table '(invocable-ks-table hi ))))
    (loop for poss-ks being the hash-elements of invocables
          with-key key
          do
      (princ key))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;******* BBDISP.LISP ********************************************
;******* Last-edit 05/28/87 *************************************

;;; This file, bbdisp.lisp, contains the variables, flavors, and methods
; which produce the display for the Blackboard Architecture.  The color
; map definition is from Mort's code.
;;;


;;; A fine-grained gray scale with some colors for marking things at the high end.

(DEFUN gray-lookup (index)
  (LET ((val (+ 3 (* 4 index))))
    (VALUES val val val)))

(DEFCONSTANT *gray-map*
  (LET ((map 0))
    (LOOP FOR index FROM 0 TO 248
          DO
          (MULTIPLE-VALUE-BIND (r g b) (gray-lookup index)
            (SEND COLOR:COLOR-SCREEN :WRITE-COLOR-MAP index r g b map)))
    ;; add color markers at the high end of the scale
    (SEND COLOR:COLOR-SCREEN :WRITE-COLOR-MAP 249 500 0 500 map)
    (SEND COLOR:COLOR-SCREEN :WRITE-COLOR-MAP 250 1023 0 0 map)
                                                              ;red
    (SEND COLOR:COLOR-SCREEN :WRITE-COLOR-MAP 251 1023 1023 0 map)
                                                              ;yellow
    (SEND COLOR:COLOR-SCREEN :WRITE-COLOR-MAP 252 0 1023 0 map)
                                                              ;green
    (SEND COLOR:COLOR-SCREEN :WRITE-COLOR-MAP 253 0 1023 1023 map)
                                                              ;cyan
    (SEND COLOR:COLOR-SCREEN :WRITE-COLOR-MAP 254 0 0 1023 map)
                                                              ;blue
    (SEND COLOR:COLOR-SCREEN :write-color-map 255 1023 1023 1023 map)
                                                              ;white
    map)
  "gray-scale map for color monitor")

(DEFVAR *draw-color* 255)

(DEFVAR *danger-color* 250
"the no-clear-path condition is signaled by a red bar")

(DEFVAR *line-color* 251
"abstract features derived from the road data are shown as yellow lines")

(DEFVAR *path-color* 252
"computed clear path is marked in green")

(DEFVAR *edge-color* 254
"texture features are marked in blue")

(DEFVAR *bar-graph-back* 249)

(DEFVAR *demo-window-alu*
        (SEND COLOR:COLOR-SCREEN :COMPUTE-COLOR-ALU COLOR:ALU-X 0))


;;;a flavor for invocable and active KSs (bar graphs)
;;;
```

```
(DEFFLAVOR bar-window
        ((num-elems 0))                                 ;height of bar graph
        ( TV:WINDOW)
  (:CONC-NAME NIL)
  (:WRITABLE-INSTANCE-VARIABLES num-elems))


;;;a flavor for the marked-images display
;;;

(DEFFLAVOR flag-window
        ((right-missing 25)
         (left-missing 55)
         (left 50))
        (TV:WINDOW)
  (:CONC-NAME NIL)
  (:WRITABLE-INSTANCE-VARIABLES))

;;;a flavor for the KS processes and results display
;;;

(DEFFLAVOR sensitive-icon-display
        ()
        (DW:DYNAMIC-WINDOW))

;;;INCR-GRAPH and DECR-GRAPH
; These methods are used to update the invocable and active KS bar
; graphs.
;;;

(DEFMETHOD (incr-graph bar-window) ()
  (LET* ((curr-num (num-elems SELF))
         (position (- 60 (* 15 curr-num)))
         (block
           (TV:MAKE-SHEET-BIT-ARRAY
             SELF 32 15                                 ;make the bar increment
             :INITIAL-VALUE 7)))
    (SEND SELF :BITBLT
          TV:ALU-XOR 45 15 block 0 0 0 position)
    (SETF (num-elems SELF) (+ curr-num 1))))            ;update the number of elements

(DEFMETHOD (decr-graph bar-window) ()
  (LET* ((curr-num (num-elems SELF))
         (position (- 60 (* 15 (- curr-num 1))))
         (block
           (TV:MAKE-SHEET-BIT-ARRAY
             SELF 32 15
             :INITIAL-VALUE 7)))
    (SEND SELF :BITBLT                                  ;xor to erase a block
          TV:ALU-XOR 45 15 block 0 0 0 position)
    (SETF (num-elems SELF) (- curr-num 1))))            ;update the number of elements

(defmethod (flag-condition flag-window) (cond-string x y alu)
    (send self :draw-string cond-string x y (+ 900 x) y
          nil (si:backtranslate-font 'fonts:eurex24i) alu))


(defmethod (erase-condition flag-window) (text-base-line)
  (let ((win-height 25))
    (send self :draw-rectangle tv:width (+ 5 win-height)
          0 (- text-base-line win-height) tv:erase-aluf)))

;;;MAKE-MAIN-BBDISP-WINDOW
```

```
; This function creates and labels the main display window.  All other
; BA display windows are inferior to this one.
;;;

(DEFUN make-main-bbdisp-window
       (&REST options
        &KEY (superior (COLOR:FIND-COLOR-SCREEN :CREATE-P T))
        &ALLOW-OTHER-KEYS)
  (APPLY #'TV:MAKE-WINDOW 'TV:WINDOW
         :BLINKER-P NIL
         :BORDERS 2
         :SAVE-BITS NIL
         :EXPOSE-P T
         :LABEL '(:TOP
                  :STRING "BLACKBOARD  ARCHITECTURE"
                  :FONT FONTS:DUTCH20BI )
         :EDGES '(0 0 1000 1023)
         :SUPERIOR superior
         options))

;;;CREATE-ICON-STREAM
; This function creates the window *icon-disp*.  It is a dynamic window,
; with vertical scrolling permitted.
;;;

(DEFUN create-icon-stream ()
  (DECLARE (SPECIAL *bbdisp* *icon-disp*))
  (SETQ *icon-disp*
        (TV:MAKE-WINDOW 'sensitive-icon-display
                        :SUPERIOR *bbdisp*
                        :INSIDE-HEIGHT 600
                        :INSIDE-WIDTH  940
                        :EXPOSE-P T
                        :SAVE-BITS NIL
                        :BLINKER-P NIL
                        :TOP 200
                        :LEFT 25
                        :MARGIN-COMPONENTS
                                              ;label and margin options
                        '((DW:MARGIN-BORDERS
                            :THICKNESS 1)
                          (DW:MARGIN-WHITE-BORDERS
                            :THICKNESS 2)
                          (DW:MARGIN-LABEL
                            :MARGIN :TOP
                            :CENTERED-P NIL
                            :BOX :INSIDE
                            :EXTEND-BOX-P T
                            :STYLE (:DUTCH :ITALIC :VERY-LARGE)
                            :STRING "Time VS Action Graph")
                          (DW:MARGIN-SCROLL-BAR
                            :VISIBILITY :NORMAL)))))

;;;MAKE-BAR-GRAPH-WINDOW
; This function creates the two windows used to display the invocable and
; the active KSs.
;;;

(DEFUN make-bar-graph-window (left top name)
  (DECLARE (SPECIAL *bbdisp*))
  (SEND *demo-window-alu* :SET-FILL-DATA *bar-graph-back*)
  (TV:MAKE-WINDOW 'TV:WINDOW
                  :SUPERIOR *bbdisp*
```

```
                    :INSIDE-HEIGHT 100
                    :INSIDE-WIDTH  450
                    :BORDERS 2
                    :EXPOSE-P T
                    :SAVE-BITS NIL
                    :BLINKER-P NIL
                    :TOP top
                    :LEFT left
                    :LABEL '(:TOP
                              :STRING ,name
                              :FONT FONTS:DUTCH20BI)
                    :ERASE-ALUF *demo-window-alu*))

;;;MAKE-BAR-WINDOW
; This function creates each of the windows for the individual bars of the
; bar graph.
;;;

(DEFUN make-bar-window (left name super)
  (SEND *demo-window-alu* :SET-FILL-DATA *bar-graph-back*)
  (TV:MAKE-WINDOW 'bar-window
                    :SUPERIOR super
                    :BOTTOM 120
                    :WIDTH 50
                    :BORDERS 0                    ;do not outline these windows
                    :EXPOSE-P T
                    :SAVE-BITS NIL
                    :BLINKER-P NIL
                    :TOP 25
                    :LEFT left
                    :LABEL '(:STRING ,name)
                    :ERASE-ALUF *demo-window-alu*))

;;;MAKE-FLAG-WINDOW
; This function makes a borderless window for use by the display KSs.  The
; window is wide enough for 3 sets of images.
;;;

(DEFUN make-flag-window ()
  (DECLARE (SPECIAL *bbdisp*))
  (TV:MAKE-WINDOW 'flag-window
                    :SUPERIOR *bbdisp*
                    :BOTTOM 1020
                    :WIDTH 950
                    :LABEL NIL
                    :BORDERS 0
                    :EXPOSE-P t
                    :SAVE-BITS NIL
                    :BLINKER-P NIL
                    :TOP 850
                    :LEFT 5))



;;;CREATE-BARS
; This function sets up the bar windows in the proper positions and assigns
; names and labels to them
;;;

(DEFUN create-bars ()
  (DECLARE (SPECIAL *active-ks-window* *imga* *imgi*
                    *inv-ks-window* *mxra* *mnra*
                    *edga* *lsda* *rsda* *mxri*
```

```
; -*- Mode: lisp; Base: 10; Package: CL-USER; Syntax: Common-Lisp -*-


(defvar *image-queue1* nil)
(defvar *image-queue2* nil)

(defconstant *image-queue-length* 5)

(defconstant *rows* 128)

(defconstant *columns* 128)


(defflavor sensor-image
        (image                                  ; the image array (128 X 128)
         (sequence 0)                           ; image sequence number
         time-tag                               ; time the image was read into
                                                ; the image array
         type                                   ; coordinate representation
                                                ; either 'xyz or 'aar
         (row-size 128)                         ; number of rows in the image array
         (col-size 128)                         ;
         (center-col 64))                       ; the middle col of the image array
        ()
  :writable-instance-variables
  :initable-instance-variables
  (:conc-name nil))


(defresource image-object (&optional (rows 128) (columns 128))
  :constructor (make-instance 'sensor-image
                              :row-size 128
                              :center-col 64
                              :image (make-array (list rows columns)
                                                 :element-type '(unsigned-byte 8)))
  :initial-copies 0)



(defun start-sensor2 (image-set1 image-set2)
  (process-run-function 'sensor-process
    #'start-sensor-process image-set1 image-set2))


(defun start-sensor-process (image-set1 image-set2)
  (declare (special *rows* *columns*))
  (let ((file-list1 (find-files image-set1))
        (file-list2 (find-files image-set2))
        (last-row (- *rows* 1))
        (last-col (- *columns* 1))
        pix
        image-obj
        (image-num -1))
    (loop for pathname in file-list1
          do
      (setq image-num (+ image-num 1))
      (setq image-obj (allocate-resource 'image-object *rows* *columns*))
      (with-open-file (image-stream pathname
                                    :direction :input
                                    :if-does-not-exist :error
                                    :characters nil
                                    :byte-size 8)
       (loop for row from 0 to last-row                    do
```

```
          (loop for col from 0 to      last-col                    do
            (setq pix (read-byte image-stream))
            (setf (aref (image image-obj) row col )
                  pix))))                                  ; write to the image array
;       (setf (time-tag image-obj) (zl:time))
      (setf (time-tag image-obj) (* 50 image-num))
      (setf (sequence image-obj) image-num)

       (push image-obj *image-queue1*)

;;;display stuff....
;
       (cond ((> (length *image-queue1*) *image-queue-length*)
              (deallocate-resource 'image-object (car (last *image-queue1*)))
              (setq *image-queue1* (butlast *image-queue1*))))
       (let ((top (* 3 (time-tag (first *image-queue1*)))))
         (multiple-value-bind (lt ignore ignore bm)
             (send *icon-disp* :visible-cursorpos-limits)
           (if (> (+ top 100) bm)
               (send *icon-disp* :set-viewport-position lt
                     (- top 300))
               )))
           (sh (image (first *image-queue1*)) 5
               (+ 5 (* 3 (time-tag (first *image-queue1*)))))
;;;end of display stuff...
;

       (let ((pathnm (pop file-list2)))
           (setq image-obj (allocate-resource 'image-object *rows* *columns*))
           (with-open-file (image-stream pathnm
                                         :direction :input
                                         :if-does-not-exist :error
                                         :characters nil
                                         :byte-size 8)
       (loop for row from 0 to last-row                    do
         (loop for col from 0 to      last-col                  do
           (setq pix (read-byte image-stream))
           (setf (aref (image image-obj) row col ) pix))); write to the image array
       (setf (time-tag image-obj) (* 50 image-num))
       (setf (sequence image-obj) image-num)
       (push image-obj *image-queue2*)
       (cond ((> (length *image-queue2*) *image-queue-length*)
              (deallocate-resource 'image-object (car (last *image-queue2*)))
              (setq *image-queue2* (butlast *image-queue2*))))
       ))))


(defun find-files (wild-card-path &optional (first 1) (last :last))
  (let* ((file-list (cdr (mapcar #'car (fs:directory-list wild-card-path :sorted))))
         (ll (length file-list)))
    (when (equal last :last) (setq last ll))
    (setq file-list (zl:firstn (min last ll) file-list))
    (setq file-list (nthcdr (max 0 (- first 1)) file-list))
    file-list))
```

```
; -*- Mode:  lisp; Base: 10; Package: CL-USER; Syntax: Common-Lisp -*-

(DEFun make-req ( data-type time-ref)
  (add-to-bb
    (MAKE-INSTANCE 'request                          ;make request object
                   :priority  7                      ;pass on priority
                   :quantity  NIL
                   :req-type  data-type              ;type of data requested
                   :requestor 'sprout                ;type of KS requesting
                   :input-key time-ref               ;unique identifier
                   :level     'hi
                   :type      'requests
                   :key       (LIST data-type time-ref ))))



;;;set up some variables we can use to look at things...
;;;


(setq lo (gethash 'lo *bb*))
(setq hi (gethash 'hi *bb*))
(setq ikt (gethash 'invocable-ks-table hi))
(setq req (gethash 'requests hi))
(setq raw (gethash 'image lo))
(setq mxrow (gethash 'max-row lo))
(setq mnrow (gethash 'min-row lo))
(setq ledge (gethash 'left-edge lo))
(setq redge (gethash 'right-edge lo))
(setq in-op (gethash 'have-done hi))
(setq lside (gethash 'left-side lo))
(setq rside (gethash 'right-side lo))

(defun init-bba ()
  (declare (special ikt req raw mxrow mnrow
                    ledge redge lside rside
                    in-op  *icon-disp*
                    *flag-window*))
  (clrhash ikt)
  (clrhash req)
  (clrhash raw)
  (clrhash mxrow)
  (clrhash mnrow)
  (clrhash ledge)
  (clrhash in-op)
  (clrhash lside)  (clrhash rside)
  (clrhash redge)
  (send *icon-disp* :clear-history)
  (send *flag-window* :clear-window))

(defun sense ()
  (start-sensor2
                "vino:>images>tacom-cc3>*.*" "vino:>images>tacom-dh3>*.*"))

(defun start-the-madness ()
  (declare (special ikt *mxri* *mnri* *edgi*
                    *lsdi* *rsdi* *imgi* *flgi*
                    *gci* ))
  (loop until nil
        doing
    (send *demo-window-alu* :set-fill-data *bar-graph-back*)
    (send *imgi* :clear-window)
```

```
(setf (num-elems *imgi*) 0)
(send *mxri* :clear-window)
(setf (num-elems *mxri*) 0)
(send *demo-window-alu* :set-fill-data *bar-graph-back*)
(send *mnri* :clear-window)
(setf (num-elems *mnri*) 0)
(send *edgi* :clear-window)
(setf (num-elems *edgi*) 0)
(send *demo-window-alu* :set-fill-data *bar-graph-back*)
(send *lsdi* :clear-window)
(setf (num-elems *lsdi*) 0)
(send *rsdi* :clear-window)
(setf (num-elems *rsdi*) 0)
(send *demo-window-alu* :set-fill-data *bar-graph-back*)
(send *flgi* :clear-window)
(setf (num-elems *flgi*) 0)
(send *gci* :clear-window)
(setf (num-elems *gci*) 0)

(check-emergency-cases)
(loop for i being the hash-elements of ikt          ;invoke all emergency cases
            with-key k
      doing
  (invoke (eval(eval (first k))) i))
(clrhash ikt)
(check-flags-and-monitors)
(loop for i being the hash-elements of ikt          ;invoke all flags and monitors
            with-key k
      doing
  (invoke (eval(eval (first k))) i))
(clrhash ikt)
(check-every-other-ks)
(loop repeat 1
      for i being the hash-elements of ikt
            with-key k
      doing
  (invoke (eval(eval (first k))) i))
(clrhash ikt)))
```

```
                    *mnri* *edgi* *rsdi* *lsdi*
                    *flga* *flgi* *gca* *gci*
                    *clra* *clri*))
  (SETQ *imga* (make-bar-window 3 "IMG" *active-ks-window*))
  (SETQ *mxra* (make-bar-window 53 "MXR" *active-ks-window*))
  (SETQ *mnra* (make-bar-window 103 "MNR" *active-ks-window*))
  (SETQ *edga* (make-bar-window 153 "EDG" *active-ks-window*))
  (SETQ *rsda* (make-bar-window 203 "RSD" *active-ks-window*))
  (SETQ *lsda* (make-bar-window 253 "LSD" *active-ks-window*))
  (SETQ *flga* (make-bar-window 303 "FLG" *active-ks-window*))
  (SETQ *gca* (make-bar-window 353 " GC" *active-ks-window*))
  (setq *clra* (make-bar-window 403 "CLR" *active-ks-window*))
  (SETQ *imgi* (make-bar-window 3 "IMG" *inv-ks-window*))
  (SETQ *mxri* (make-bar-window 53 "MXR" *inv-ks-window*))
  (SETQ *mnri* (make-bar-window 103 "MNR"  *inv-ks-window*))
  (SETQ *edgi* (make-bar-window 153 "EDG"  *inv-ks-window*))
  (SETQ *rsdi* (make-bar-window 203 "RSD"  *inv-ks-window*))
  (SETQ *lsdi* (make-bar-window 253 "LSD"  *inv-ks-window*))
  (SETQ *flgi* (make-bar-window 303 "FLG" *inv-ks-window*))
  (SETQ *gci* (make-bar-window 353 "GC" *inv-ks-window*))
  (setq *clri* (make-bar-window 403 "CLR" *inv-ks-window*)))

;;;SH
; This function is called to display an image to a specified position in
; the icon-display window (*icon-disp*).
;;;

(DEFUN sh (it x y)
  (DECLARE (SPECIAL *img-disp*))
  (SEND  *icon-disp*
         :BITBLT TV:ALU-SETA (CAR (ARRAY-DIMENSIONS it))
         (CADR (ARRAY-DIMENSIONS it)) it 0 0 x y))

;;;The following expressions just use the above functions to set up the
; display.
;;;

(SETQ *bbdisp* (make-main-bbdisp-window))

(SETQ *active-ks-window*
      (make-bar-graph-window 25 50 "Active Knowledge Sources"))

(SETQ *inv-ks-window*
      (make-bar-graph-window 520 50 "Invocable Knowledge Sources"))

(SETQ *icon-disp*
      (create-icon-stream))

(setq *flag-window* (make-flag-window))

(create-bars)
```

DISTRIBUTION LIST

	Copies
Commander Defense Technical Information Center Bldg. 5, Cameron Station ATTN: DDAC Alexandria, VA 22304-9990	12
Manager Defense Logistics Studies Information Exchange ATTN: AMXMC-D Fort Lee, VA 23801-6044	2
Commander U.S. Army Tank-Automotive Command ATTN: ASQNC-TAC-DIT (Technical Library) Warren, MI 48397-5000	2
Commander U.S. Army Tank-Automotive Command ATTN: AMSTA-CF (Dr. K. Oscar) Warren, MI 48397-5000	1
Director U.S. Army Materiel Systems Analysis Activity ATTN: AMXSY-MP (Mr. Cohen) Aberdeen Proving Ground, MD 21005-5071	1
Commander U.S. Army Tank-Automotive Command ATTN: AMSTA-RV (L. Sieh) Warren, MI 48397-5000	3
Commander U.S. Army Tank-Automotive Command ATTN: AMSTA-ZTR (J. Lane) Warren, MI 48397-5000	3